Population-based harmony search using GPU applied to protein structure prediction

Marlon H. Scalabrin*

Electrical Engineering and
Industrial Informatics Post-Graduate Programme,
Federal University of Technology – Paraná (UTFPR),
Av. 7 de setembro, 3165 – 80230-901 Curitiba, Brazil
Fax: +55-41-3310-4683
E-mail: marlonscalabrin@yahoo.com.br
*Corresponding author

Rafael S. Parpinelli

Department of Computer Science,
Santa Catarina State University (UDESC),
Campus Universitário Prof. Avelino Marcante s/n,
89223-100 Joinville (SC), Brazil
Fax: +55-47-4009-7940
E-mail: parpinelli@joinville.udesc.br

Cesar M.V. Benítez and Heitor S. Lopes

Electrical Engineering and
Industrial Informatics Post-Graduate Programme,
Federal University of Technology – Paraná (UTFPR),
Av. 7 de setembro, 3165 – 80230-901 Curitiba, Brazil
Fax: +55-41-3310-4683
E-mail: cesarvargasb@gmail.com
E-mail: hslopes@utfpr.edu.br

Abstract: This work presents a new evolutionary algorithm based on the standard harmony search strategy, called population-based harmony search (PBHS). Also, this work provides a parallelisation method for the proposed PBHS by using graphical processing units (GPU), allowing multiple function evaluations at the same time. Experiments were done using a benchmark of a hard scientific problem: protein structure prediction with the AB-2D off-lattice model. The performance and the solution quality were evaluated and compared using four implementations: two concerning the standard HS, one running in CPU and another running in GPU, and two implementations concerning the PBHS, also running in CPU and in GPU. Results show that the quality of solutions and speed-ups achieved by the PBHS is significantly better than the HS.

Keywords: population-based metaheuristics; harmony search; protein structure prediction; PSP; graphics processing units; GPU; compute unified device architecture; CUDA; evolutionary algorithms.

Reference to this paper should be made as follows: Scalabrin, M.H., Parpinelli, R.S., Benítez, C.M.V. and Lopes, H.S. (2014) ‘Population-based harmony search using GPU applied to protein structure prediction’, *Int. J. Computational Science and Engineering*, Vol. 9, Nos. 1/2, pp.106–118.

Biographical notes: Marlon H. Scalabrin graduated in Computer Engineering at Ponta Grossa State University in 2007 and received his MSc from the Federal University of Technology Paraná in 2012. His research interests are evolutionary computation and parallel computing.

Rafael S. Parpinelli graduated in Computer Science by the Maringá State University (1999) and received his MSc from the Federal University of Technology Paraná in 2001. Since 2004, he has been an Assistant Professor at the Santa Catarina State University in Joinville, Brazil. His research interests are evolutionary computation and bioinformatics.

Cesar M.V. Benítez graduated in Electronic Engineering and received his MSc from the Federal University of Technology Paraná in 2007 and 2010, respectively. His research interests are evolutionary computation, bioinformatics and reconfigurable computing.

Copyright © 2014 Inderscience Enterprises Ltd.

Heitor S. Lopes received his PhD in Electrical Engineering from the Federal University of Santa Catarina at Florianópolis, Brazil in 1996. Since that time, he is with the graduate programme in Electrical Engineering and Computer Science of the Federal University of Technology Paraná, Curitiba, Brazil. He is also, the Head and Founder of the Bioinformatics Laboratory. His current research interests are evolutionary computation, high-performance computing and bioinformatics.

1 Introduction

Many bioinformatics problems are featured mainly to be non-linear and strongly constrained. This is the case of the protein structure prediction (PSP) problem approached in this paper. Due to the limitations of exact methods for solving such a class of problems, the need for more robust techniques arises. Along decades, evolutionary computation (EC) and swarm intelligence (SI) have provided a large range of flexible and robust optimisation methods, capable of dealing successfully with complex optimisation problems. Both EC and SI provide population-based methods where each individual of a population represents a tentative solution to the problem to be solved. This is the case, for instance, of genetic algorithms, differential evolution (DE), particle swarm optimisation, ant colony optimisation, artificial bee colony algorithm and many other nature-inspired methods (Parpinelli and Lopes, 2011).

Algorithm 1 shows the general pseudo-code of a population-based algorithm. The main loop (between lines 3–7) represents the generational loop, and line 4 defines the mechanism or criterion for selecting the best solutions (i.e., survival of the fittest as in EC, or simply to discard the worst solutions). Two important characteristics of population-based algorithms, and also for metaheuristics in general, are intensification and diversification procedures. In line 5 of the algorithm, intensification (also called exploitation) intends to search locally and more intensively around the best solutions (i.e., by the crossover operator in genetic algorithms (GA), or a greedy search), while diversification (also called exploration) leads the algorithm to explore globally the search space (i.e., by the mutation operator in GA, or a large-scale randomisation).

Algorithm 1 General pseudo-code of a population-based algorithm

1: *Initialise* the population with random candidate solutions;
2: *Evaluate* each candidate solution;
3: **while** convergence criteria is not satisfied **do**
4: Perform competitive *selection*;
5: Apply *intensification* and *diversification* procedures;
6: *Evaluate* the new pool of candidate solutions;
7: **end while**

Source: Parpinelli and Lopes (2011)

This paper presents a new evolutionary algorithm based on the standard harmony search (HS) strategy, called population-based harmony search (PBHS). The HS is inspired by the improvisation process of a musician searching for the best harmony (Geem et al., 2001). A harmony represents the solution vector and the improvisation guides the balance between exploitation and exploration. The HS algorithm is easy to implement, and can be adopted without major modifications to solve different classes of problems.

The main drawback of these population-based techniques when facing a complex problem is the high number of function evaluations to be performed at each iteration or generation (as shown in line 6 of Algorithm 1), thus leading to a very time-consuming process. At each iteration all the current candidate solutions have to be evaluated. Since the evaluation of candidate solutions repeats the same procedure, the use of parallel processing at this point may be very useful for reducing the overall processing time of the algorithm.

In last years, the use of GPUs as platform to run population-based metaheuristics has become common in several scientific applications (Bozejko et al., 2009; Tan and Zhou, 2010; Yu et al., 2005). Graphics processing units (GPUs) are computer boards that were originally designed as a graphics-image processing devices. However, in very recent years, GPUs started to be used for general-purpose high-performance computing applications. Thanks to its parallel computing capability and fast float-point operation, GPUs can achieve impressive computing performance (Che et al., 2008). Consequently, in recent years, they have been used in many scientific and engineering applications – see, for instance, Komatitsch et al. (2010), Moorkamp et al. (2010), Shams et al. (2010) and Sunarso et al. (2010).

To foster the use of GPU boards for general-purpose computing, some platforms have been developed, such as BrookGPU (Stanford University) (Buck et al., 2004), and compute unified device architecture (CUDA, NVIDIA Corporation) (NVIDIA, 2007). These platforms have greatly simplified the programming tasks on GPU. However, taking advantage of the parallel processing capabilities of a GPU is not trivial at all, and, for a given problem, usually there are many different possibilities of implementation and memory usage.

In this work, the performance (regarding time consuming) and the solution quality are evaluated and compared using four implementations: two concerning the standard HS, one running in a central processing units (CPU) and another running in GPU, and two

implementations concerning the proposed PBHS, also running in CPU and in GPU.

This paper is organised as follows. In Section 2 a review of literature is presented, focusing on GPU computing, the standard HS algorithm, and the PSP problem. Next, Section 3.2 presents and details the proposed PBHS algorithm. The experiments are presented in Section 4. Results and their discussion are shown in Section 5. Finally, conclusions and future work are presented in Section 6.

2 Literature background

2.1 GPU-based computing

In the last three decades, processors (CPU) of desktop computers have significantly improved their performance, thus boosting by orders of magnitude the computational power of personal computers. However, since the GPUs appeared, their performance has improved at an extraordinary rate (Mohanty, 2009). This fact has driven the attention of many researchers and software developers for using GPU for general, not graphics, processing. As a consequence, GPU tends to be one of the most powerful processing technologies for scientific and engineering computing (Che et al., 2008; Owens et al., 2007).

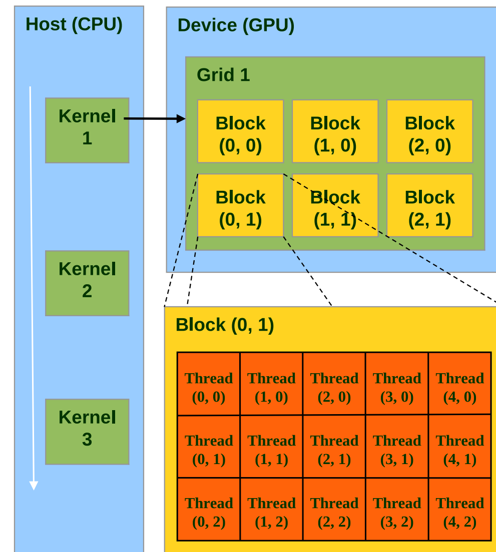
Recently, NVIDIA Corporation has developed CUDA, a platform for developing parallel computing applications directly onto GPU boards. Using this resource it is possible to use the power of parallel processing in a simple way, by using a simple extension of C programming language (Garland et al., 2008). CUDA was introduced at the beginning of 2007 as a Software Developers Kit (SDK) with compilers for Windows and Linux operating systems (NVIDIA, 2007). The current version is 3.0 meaning that it has been greatly improved along time. A programme developed using CUDA can take the advantage of the multiprocessors of a GPU board, thus increasing speed-up much faster than the current multicore CPUs (NVIDIA, 2010).

The CUDA architecture is composed, mainly, by the set of instructions CUDA instruction set architecture (ISA) and by the parallel computing hardware (multiprocessors) of the GPU board.

The execution of a task in GPU is done by function calls named 'kernels' that, in turn, initialise several identical 'threads' in the GPU. Each thread is responsible for processing a part of a large set of data (NVIDIA, 2009). This sort of parallel processing is known as SIMD – single instruction, multiple data (Parhami, 2002).

The initialisation of a kernel is accomplished by the developer and configured as needed. The distribution of threads in the GPU is also configured in the initialisation in such a way that they are grouped into 'blocks' which, in turn, are part of a computational 'grid'. CUDA uses a matrix-like distribution of the elements in the GPU. The distribution of blocks in a grid can be done in two dimensions, and the distribution of threads in a block can be done in three dimensions. Figure 1 shows an example of the spatial distribution of a grid, detailing their blocks and threads in a bi-dimensional way, as well as the access of the CPU host to the GPU device.

Figure 1 Distribution of threads and blocks in a computational grid of a GPU (see online version for colours)



Source: Based on Kirk and Hwu (2009a)

Each thread, as well as each block, has a unique index (`threadIdx` and `blockIdx`, respectively) that allows them to compute the memory index in which their corresponding data are available for processing. Consequently, the total number of threads has a direct relationship with the size of the data to be processed.

Threads of the same block can communicate each other by using specific shared memory spaces. The synchronisation of the threads of a block allows the developer to establish a common checkpoint for them. This means that only when all threads have reached such point they can proceed the execution.

The current GPUs are constructed with many multiprocessors with several cores, named streaming multiprocessors (SM). The number of threads per block is currently limited to 512 (NVIDIA, 2010), provided it does not override the available memory and shared resources in each SM. There is, also, a physical limit of 768 to the number of threads per SM. Since each block is fully allocated in a single SM, its size must be in such way to maximise the usage of computational resources of a SM (Kirk and Hwu, 2008).

Each SM creates, manages, schedules and runs the threads in groups of 32, called 'warp'. In fact only the threads in each warp are executed in parallel. All threads of a warp start the same programme at the same time, however, they run independently. If a block have a size larger than 32 it is divided into warps that are executed sequentially. The execution of a warp is more efficient when their 32 threads run simultaneously the same instruction, having the same execution flow. Conditional branch instructions in the programme operating over different data can lead to divergences in the execution

path (that is, different execution flows). In such situation they are run separately and sequentially by the warp, thus decreasing significantly the achievable performance of the system (NVIDIA, 2010).

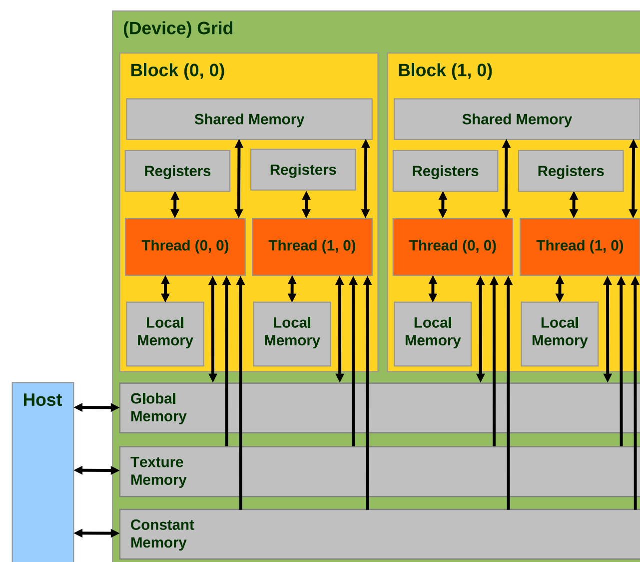
2.2 Memory usage in CUDA

In the CUDA architecture, there are six different memory types, represented in Figure 2, each one with their specific access time, permissions, scope and lifetime (Kirk and Hwu, 2008; NVIDIA, 2008, 2010):

1. *Register memory*: it is a fast-access memory in which primitive data types are stored (i.e., int, float, char). Registers have read and write permissions, scope is restricted to each thread and lifetime is equivalent to its thread.
2. *Local memory*: it is the memory in which addressable variables are stored, such as pointers to variables in the global memory, and vectors. Primitive data types are allocated in local memory when the register memory is full. This memory has the same permissions and scope of the registers.
3. *Shared memory*: it allows the information exchange between threads running in the same block. This memory has fast access, and read and write permissions. The lifetime is the same of its block.
4. *Global memory*: it has read and write permissions and global scope. It has a slow-access time, and the values stored in it are not dependent of the execution time of the kernels. This means that it can store information to be shared by different kernels or different runs of the same kernel. This memory is also used to share data between the host (CPU) and the board (GPU). It can be allocated and released anytime during the execution of the programme.
5. *Constant memory*: it has slow access time, permissions and scope similar to the global memory. Once the values are stored they cannot be changed during the execution of the programme. In current GPU boards, the constant memory space is limited to 64 Kbytes.
6. *Texture memory*: it is similar to the constant memory, except by the fact that it has an automatic caching mechanism.

In the development of applications with CUDA there are many different possibilities of using the GPU resources (blocks, threads and memories). Depending on how these resources are used, different performances can be achieved in the execution of a kernel (Kirk and Hwu, 2008). Hence, the way the parallelism is implemented by adjusting the size of the blocks and threads affect directly the performance of kernel execution.

Figure 2 Access and scope of memories (see online version for colours)



Source: Based on Kirk and Hwu (2009b)

Two other important development features that affect directly the performance are the number of kernel calls, and the memory usage. The first feature increases the communication overhead between CPU and GPU board proportionally to the the number of kernel calls. The second feature regards to use the fastest memories such as local and shared memories always when it is possible.

2.3 Harmony search

The HS metaheuristic is inspired by musician skills of composition, memorisation and improvisation. As musicians use their skills to pursue the perfect composition with a perfect harmony, the HS algorithm use its search strategies to pursuit for the optimum solution of an optimisation problem. Some successful applications of the HS algorithm can be found in Fesanghary et al. (2008), Geem et al. (2001), Geem (2009, 2010), Saka (2007) and Lee and Geem (2004).

The pseudo-code of the HS algorithm is presented in Algorithm 2 (Geem et al., 2001). The HS algorithm starts with a harmony memory (HM) of size HMS, in which each position is occupied by a harmony of size N (musicians). At each step of improvising a new harmony is generated from the harmonies already present in the harmony memory. If the new harmony generated is better than the worst harmony in the harmony memory, this is replaced with the new one. The steps to improvise and update the harmony memory are repeated until the maximum number of improvisations (MI) is achieved.

Algorithm 2 Pseudo-code of the HS algorithm

```
1: Parameters: HMS, HMCR, PAR, MI, FW
2: Start
3: Objective Function f(x⃗), x⃗ = [x_1, x_2, ..., x_N]
4: Initialise HM: x^i, i = 1, 2, ..., HMS
5: Evaluate each Harmony in HM: f(x^i)
6: cycle ← 1
7: while cycle < MI do
8:    for j ← 1 to N do
9:       if random ≤ HMCR then {Rate of Memory Consideration}
10:         x'_j ← x^i_j, with i ∈ [1, HMS] {chosen randomly}
11:         if random ≤ PAR then {Pitch Adjusting Rate}
12:            x'_j ← x'_j ± r × FW {with r random}
13:         end if
14:      else {Random Selection}
15:         Generate x'_j randomly
16:      end if
17:   end for
18:   Evaluate new harmony generated: f(x')
19:   if f(x') is better than worst harmony in HM then
20:      Update Harmony Memory
21:   end if
22:   cycle ← cycle + 1
23: end while
24: Results and views
25: End
```

As shown in Geem et al. (2001) and Mahdavi et al. (2007), the algorithm can be described by five main steps, detailed below[1]:

1. *Initialisation of problem and algorithm parameters*: In the first step, as in any optimisation problem, the problem is defined as an objective function to be optimised (line 3 of Algorithm 2), which can or cannot be constrained. Originally, HS was designed for solving minimisation problems (Geem et al., 2001).

   In this step the parameters of the algorithm are also defined. The four main parameters are: the harmony memory size – HMS, the harmony memory consideration rate – $HMCR$, the pitch adjusting rate – PAR, and the maximum number of improvisations (maximum improvisation – MI).

2. *Harmony memory initialisation*: The second step is to initialise the harmony memory (line 4) with a number of randomly generated harmonies. The HM is the vector in which the best harmonies found during run are stored. Each harmony is a vector representing a possible solution to the problem.

3. *Improvise a new harmony*: In the third step, a new harmony is improvised based on the harmonies that currently exist in HM, and the new harmony is a combination of several other harmonies (between lines 8–17). For each variable of the new harmony, a harmony of the HM is arbitrarily selected by checking the probability of this value to be or not to be used ($HMCR$). If another harmony is used, the value of this variable will have small adjustments (Fret width – FW) according to a probability (PAR). If it is not used the value of another harmony, a random value within the range of allowed values is assigned. Thus, the parameters $HMCR$ and PAR are responsible for establishing a balance between exploration and exploitation in the search space. The higher the $HMCR$ parameter, the more exploitation is accomplished.

4. *Update harmony memory*: In the fourth step, each new harmony just improvised is checked to see if it is better than the worst harmony from HM (lines 19–21). If this condition is confirmed, the new harmony replaces the worst one in the HM.

5. *Checking of the stopping criterion*: In the fifth step, the end of each iteration is checked to discover if the best harmony meets the stopping criterion, usually a maximum number of improvisations (MI). If so, the execution is completed. Otherwise, it returns to the second step until reaching the stopping criterion.

2.4 Protein structure prediction

Proteins are the basic structures of all living beings (Hunter, 1993). They are composed by a chain of amino acids that are linked together by means of peptide bonds. Each amino acid is characterised by a central carbon atom (also called as alpha carbon – Cα) to which are attached a hydrogen atom, a carboxyl group (COOH), an amino group (NH_2)

and a side-chain. Its known that the side-chain defines the physical and chemical properties of the amino acid (Cooper, 2000). Peptide bonds are formed from the condensation of two amino acids, when the carboxyl group of an amino acid reacts with the amino group of the other. This process is also called as dehydration because it releases a molecule of water.

Several amino acids exist in nature, but only 20 are proteinogenic. They can be classified into two classes, according to their affinity to water: hydrophilic (or polar) and hydrophobic. According to this behaviour, one can conclude that the polarity of the side chain governs the process of forming protein structures (Lodish et al., 2000).

From the chemical point of view, proteins are structurally complex and functionally sophisticated molecules (Alberts et al., 2002). The structural organisation of proteins is commonly described into four levels of complexity: primary, secondary, tertiary and quaternary structures. Its important to know that the upper levels cover the properties of lower ones. The primary structure refers to the linear sequence of amino acids, the secondary represent local conformations of some part of a three-dimensional structure. The tertiary structure represents the conformation of a polypeptide chain, i.e., the three-dimensional arrangement of the amino acids. Finally, regular associations of three-dimensional structures constitutes quaternary structures.

The *protein folding* is the process by which polypeptide chains are transformed into compact structures that perform biological functions. These functions include control and regulation of essential chemical processes for the living organisms. Under physiological conditions, the most stable three-dimensional structure is called the native conformation and actually allows a protein to perform its function.

Failure to fold into the intended three-dimensional conformation usually leads to proteins with different properties that simply become inactive. In the worst case, such misfolded (incorrectly folded) proteins can be harmful to the organism. For instance, several diseases such as Alzheimer's disease, cystic fibrosis and some types of cancer, are believed to result from the accumulation of misfolded proteins.

Its known that better understanding of the protein folding process can result in important medical advancements and development of new drugs. Thanks to the several genome sequencing projects being conducted in the world, a large number of new proteins have been discovered. However, only a small amount of such proteins have their 3-dimensional structure known. For instance, as in April/2011, the UniProtKB/TrEMBL repository of protein sequences has currently around 14 million records[2], and the Protein Data Bank – PDB (Berman et al., 2000) has the structure of only 72,386 proteins[3]. This fact is due to the cost and difficulty in unveiling the structure of proteins, from the biochemical point of view.

Computer science has an important role here, proposing models for studying the PSP problem (Lopes, 2008). Nowadays, the simulation of computational models that take into account all the atoms of a protein is frequently unfeasible, even with the most powerful computational resources. Consequently, several simplified models that abstract the protein structure have been proposed. Basically, there are two types of representation of polypeptides, the analytical and the discrete. The analytical representation describes all the information about the atoms that compose the proteins. On the other hand, the discrete representation describes a protein in a reduced level of details. Although such discrete models are not realistic, they use some biochemical properties of amino acids, and its simulation can show some interesting characteristics of real proteins. They also allow an extensively exploration of the conformational space and can be generators of hypotheses that cannot be obtained by other approaches, but that may be reproducible experimentally or through refined simulations (Dill, 1999). This is an important motivation for developing computational methods for predicting the structure of proteins. The simplest computational model for the PSP problem is known as hydrophobic-polar (HP) model, both in two (2D-HP) and three (3D-HP) dimensions (Dill et al., 1985). Although simple, the computational approach for searching a solution for the PSP using the HP models was proved to be NP-complete (Atkins and Hart, 1999; Berger and Leighton, 1998; Crescenzi et al., 1998).

Many approaches to solve the PSP were proposed, each addressing the problem by using a computational method to obtain optimal or, more frequently, quasi-optimal solutions. From the chemical point of view, the most realistic method is called molecular dynamics (also known as *ab initio*) (Hardin et al., 2002). The main idea of this approach is to simulate atom movements according to rules of classical mechanics. On the other hand, algorithms using the simplest model for the PSP were proved to be NP-complete. Therefore, this fact has motivated the development of several metaheuristics to deal with the problem.

For instance, Benítez and Lopes (2010) present an hierarchical parallel genetic slgorithm applied to the PSP using the three-dimensional HP side-chain model (3DHP-SC). An ant colony optimisation algorithm (ACO) for the PSP using both 2D and 3DHP models was presented by Shmygelska and Hoos (2005). Local search methods such as Monte Carlo, tabu search and hill-climbing were used as genetic operators for genetic algorithms by Cox et al. (2004), Jiang et al. (2003) and Tantar et al. (2007), respectively.

The DE algorithm was used by both Bitello and Lopes (2007) and Kalegari and Lopes (2010), for tacking the PSP using the 2DHP and 2D-AB off-lattice models, respectively. Possibly, these are the only works published to date using the DE algorithm for the PSP. Artificial immune systems (AIS) were applied to the PSP using the 2D and 3DHP models by Cutello et al. (2005). Also, Almeida et al. (2007) present an hybrid AIS with tabu search and a inference fuzzy system. In this work, fuzzy operator decides which antibodies will be removed from the population after the selection procedure, and the tabu search is used to define a mechanism affinity maturation of antibodies.

Other interesting methods were used to solve the PSP. For instance, Ostrovsky et al. (2001) describe a cellular automata for polymer simulation including implications for protein folding and Yanikoglu and Erman (2002) present a solution using self-organising networks neural networks (SOM) with the 2DHP model. Stillinger et al. (1993), Irback et al. (1997) and Torcini et al. (2001) employed neural networks, Monte Carlo search and biologically inspired methods using the 2D-AB off-lattice model. An extended three-dimensional version of the 2D-AB was presented by Hsu et al. (2003). Recently, Zhang and Cheng (2008) introduced an improved implementation of tabu search with the 3D-AB *off-lattice* model, obtaining good performance.

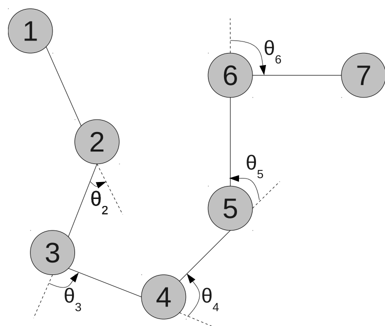
2.4.1 The AB off-lattice model

The AB off-lattice model, introduced by Stillinger and Head-Gordon (1995), was one of the first models to represent protein structures. In this model the protein sequences are composed by only two species of monomers (ξ): 'A' for hydrophobic amino acids and 'B' for hydrophilic (or polar) amino acids. Although it is a very simplified representation of a real protein structure, this model is useful to verify some of the properties of proteins in the real world.

Monomers have an unit length of distance between them, in such a way that a monomer is connected to the next one in the chain through a bond that forms an angle relative to its predecessor.

In the AB model, a protein composed of n-monomers needs $n - 2$ angles to be represented. These angles are defined in the range $[-180^o..180^o]$. Figure 3 shows an example of a hypothetical protein with seven amino acids.

Figure 3 Generic representation of a hypothetic protein structure



The model defines the energy values for the monomers: 'A' has energy 1 and 'B' has energy –1. Considering two generic monomers i and j, and the species ξ_i and ξ_j, respectively, the interaction between the monomers leads to different values of potential energy (C). Positive values represent attraction and negative, repulsion: AA bonds have energy 1 (the monomers AA tend to attract each other strongly), BB bonds have energy 1/2 (they tend to attract each other weakly) and AB or BA bonds have energy –1/2 (they have a weak repulsion). The energy of the structure of a protein with n monomers (n-mers) is given by equation (1):

$$\phi(\theta, \xi) = \sum_{i=1}^{n-1} V_1(\theta_i) + \sum_{i=1}^{n-2} \sum_{j=1+2}^{n} V_2(d_{ij}, \xi_i, \xi_j) \tag{1}$$

Equation (1) postulates two types of intermolecular potential energies, terms V_1 and V_2. The former represents the backbone potentials. It is defined by equation (2) and depends only on the angle between monomers. The latter, defined by equation (3), represents the potential energy present in the non-bonded interactions and it is known as the Lennard-Jones potential.

$$V_1(\theta_i) = \frac{1}{4}.(1 - cos(\theta_i)) \tag{2}$$

$$V_2(d_{ij}, \xi_i, \xi_j) = 4.\left(d_{ij}^{-12} - C.(\xi_i, \xi_j).d_{ij}^{-6}\right) \tag{3}$$

where

$$C(\xi_i, \xi_j) = \frac{1}{8.(1 + \xi_i + \xi_j + 5.\xi_i.\xi_j)} \tag{4}$$

Equation (4) is the potential energy due to the interaction between monomers i and j, and d_{ij} is the distance between these monomers in the chain, such that $i < j$.

3 Methodology

Section 3.1 describes the development of the HS algorithm using GPU and Section 3.2 describes the proposed PBHS.

3.1 HS using GPU

In the implementation of HS in GPU, all steps of the algorithm are executed in the GPU. This allows quick access to data in the memory, avoiding data transfer between memories and the host device. In fact, the host CPU is responsible only for the initialisation of parameters, allocation of the GPU global memory and control of iterations. Some data are also copied to the GPU, which are used during execution of the HS, i.e., upper and lower bounds, and random number generator configuration. Such approach takes advantage of the implicit parallel nature of some steps of the HS algorithm, namely, the treatment of the variables of a new harmony.

The HM initialisation is performed with one kernel call to each memory location of HM, producing a new harmony for each position. Each variable of the new harmony is independent of the others. In GPU, the HM is allocated in the global memory and it is represented linearly, for every N positions there is a different harmony.

Once the HM is loaded with the initial harmonies, the iterative process of optimisation of the HS algorithm is started. At each iteration cycle, two kernel calls are performed. The first performs the improvisation of a new harmony and the second one updates the HM.

In the improvisation step, the process of selection of each variable of the new harmony is performed

independently. In the kernel of improvisation each block, with only one thread, is responsible for one variable of the new harmony.

The HM is kept ordered, from the best harmony, at the first position, to the worst one, at the last position. Thus, updating the HM is accomplished by inserting a new harmony in its proper position. The following positions are shifted, discarding the worst. To find the insertion point, a sequential scan of the HM is done. Each variable to be replaced is treated independently.

Once the optimisation process is completed, the data relating to the best harmony found is transferred from the device to the host in order to be used and displayed.

3.2 Population-based harmony search

Aiming at improving the performance of execution of HS using parallel architectures, we developed a PBHS. Basically, PBHS includes a population of temporary harmonies, named musical arrangements (MA). In this case, MA can be considered as an extension of the HM. Such positions are included in the HM and are renewed at each cycle of the algorithm.

During the improvisation step, the whole set HM and MA are considered, in such a way that all these memory positions can be used to compose new harmonies. At the end of each cycle, among the harmony found in the MA, only the best is effectively inserted in the HM.

The implementation of PBHS has some changes that allow the algorithm to process several new harmonies at the same time. Algorithm 2 shows the differences between HS and PBHS highlighting the changes proposed in this work.

In line 4, the memory positions for the MA are jointly initialised with the HM, so that values assigned to these positions can be used to generate new valid harmonies.

In Improvisation (between lines 7–19), the population size is added to the HM size, and the positions of the population are included in the composition of new harmonies. Each thread of block of the Improvisation kernel is responsible for a different new harmony (equivalent to line 8), thus the number of blocks is equal to the population size, and each block is responsible for one position on all new harmonies (equivalent to line 9).

During the improvisation process, new harmonies are created from harmonies selected using a stochastic tournament selection procedure (Blickle, 2000) (line 11). Its important to know that this procedure tends to be less elitist that other popular selection methods (such as the roulette wheel method used in genetic algorithms). The new improvised harmonies are included in the space dedicated to the MA. Once the new harmonies are created, the evaluation of all of them are performed simultaneously, i.e., in parallel. Finally, the best harmony of the MA is selected to be used in the updating process.

In the PBHS approach, the population size has a key role. Improving the diversity of harmonies used to compose other new harmonies, the speed of convergence tends to decrease, but, on the other hand, the exploration of the search space tends to be more efficient.

4 Experiments

All experiments reported in this session were run in a personal computer with an Intel processor (Core2-Quad running at 2.8 GHz) and a NVIDIA GeForce GTX 280 graphic card, running Linux. The applications were developed in the C programming language. To implement the parallel approaches in GPU, we used CUDA[4].

The focus of the experiments was to compare both GPU implementations, measuring processing time and the solution quality, identifying the possible improvements that the use of a population can bring to the original HS algorithm.

4.1 Benchmark sequences

In the experiments reported below, a total of five synthetic protein sequences were used. These sequences have been previously used by other researchers (for instance, Hsu et al., 2003; Kalegari and Lopes, 2010). In Table 1, N is the number of monomers of the sequences (13, 21, 34 and 55 amino acids-long sequences).

Table 1 Benchmark sequences for the 2D-AB *off-lattice* model

N	*Sequence*
13	ABBABBABABBAB
21	BABABBABABBABBABABBAB
34	ABBABBABABBABBABABBABABBABBABABBAB
55	BABABBABABBABBABABBABABBABBABABBAB BABABBABABBABBABABBAB

4.2 Parameters adjustment

A factorial experiment (see Box et al., 2005 for more information) was done to adjust all the parameters of the algorithms, including the basic parameters of HS and PBHS (harmony memory consideration rate – $HMCR$ and the pitch adjusting rate – PAR). Parameters were tested in steps of 5%, in the following range: $HMCR = [75\%..95\%]$ and $PAR = [5\%..35\%]$. The combination of possible values for these parameters yields a total of 24 different experiments. Each experiment was run 30 times with different initial random seeds. The basic parameters of the HS corresponding to the best experiment are shown in Table 2. The remaining parameters were configured based on the literature. For instance, the Harmony Memory (HM), the Fret Width (FW) and the number of fitness evaluations were empirically set to 20, 15 and 5,000,000 respectively. Its important to point that the FW was exceptionally set fixed to 30 for the 34-amino acids long sequence.

Specifically for the PBHS approach, the size of MA was set to 32. This number was set to take advantage of the

Table 2 Configuration parameters of HS and PBHS, obtained by a factorial experiment

N	*HS*		*PBHS*	
	HMCR	*PAR*	*HMCR*	*PAR*
13	95%	20%	90%	20%
21	90%	5%	90%	10%
34	95%	5%	95%	10%
55	95%	5%	95%	5%

GPU features, enabling the maximum number of threads of a warp (see Section 2.1). The tournament size of the selection procedure was set to five harmonies.

The GPU implementation for the PSP uses one independent block for each aminoacid. This means that for a sequence of N aminoacids, N cores will be employed for computation.

5 Results and analysis

Due to the stochastic nature of the algorithms tested, for each benchmark sequence, both HS and PBHS were executed for 30 independent runs using different random seeds.

Regarding quality, results are shown in Table 3. In this table, the first column identifies the sizes of amino acids sequences; the second shows the currently known global minimum; the third column shows the average and standard deviation obtained by the standard HS; the fourth column shows the best value found by HS; the fifth column shows the relationship of the best solution found by HS regarding the known global minimum; the sixth, seventh and eighth columns are equivalent to the third, fourth, fifth columns, but now for PBHS.

Table 3 Comparative quality of solution

N	*Global minimum*	*HS*			*PBHS*		
		Avg fit.	*Best fit.*	*% Global minimum*	*Avg fit.*	*Best fit.*	*% Global minimum*
13	–3.29	–2.31 ± 0.70	–3.19	3.0%	–3.00 ± 0.30	–3.28	0.3%
21	–6.19	–3.76 ± 0.47	–5.00	19.2%	–4.02 ± 0.61	–5.96	3.7%
34	–10.70	–5.68 ± 0.87	–7.63	28.6%	–5.49 ± 0.98	–8.33	22.1%
55	–18.51	–6.74 ± 1.19	–9.01	51.3%	–8.18 ± 1.34	–11.51	37.8%

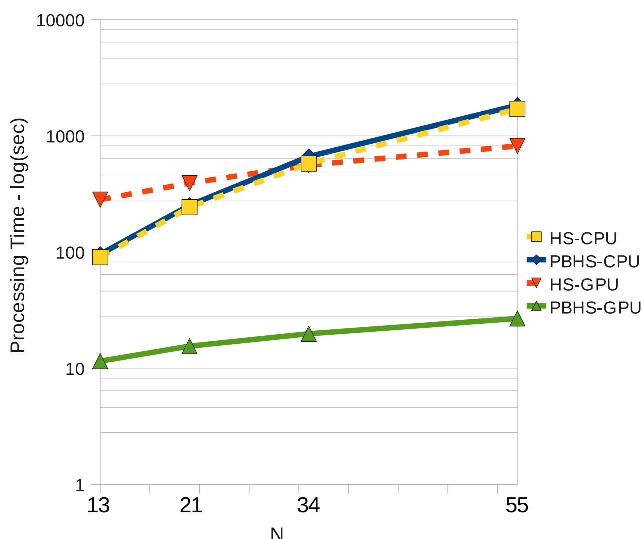
Algorithm 3 Pseudo-code of the PBHS

```
1: Parameters: HMS, HMCR, PAR, MI, FW, PS
   COMMENTPS=Population Size
2: Start
3: Objective Function f(x⃗),   x⃗ = [x_1, x_2, ..., x_N]
4: Initialise Harmony Memory and Musical Arrangements x^i, i = 1, 2, ...,(HMS+PS)
5: Evaluate each Harmony in HM: f(x^i)
6: cycle ← 1
7: while cycle < MI do
8:    for new ← 1 to PS do
9:       for j ← 1 to N do
10:         if random ≤ HMCR  then {Rate of Memory Consideration}
11:            x_j^(HMS+new) ← x_j^i, with i in [1,(HMS+PS)] {i selected by stochastic tournament}
12:            if random ≤ PAR then {Pitch Adjusting Rate}
13:               x_j^(HMS+new) ← x_j^(HMS+new) ± r × FW  {with r random}
14:            end if
15:         else {Random Selection}
16:            Generate x_j^(HMS+new) randomly
17:         end if
18:      end for
19:   end for
20:   Evaluate all new harmonies generated: f(x^i), i = HMS + (1, 2, ..., PS)
21:   Select best new harmony k generated in Musical Arrangements
22:   if f(x^k) is better than worst harmony in HM then
23:      Update Harmony Memory
24:   end if
25:   cycle ← cycle + 1
26: end while
27: Results and views
28: End
```

It is possible to observe that the PHS approach presented better solutions for all sequences, considering not only the averages, but also, the best results.

Analysing now the performance, Figure 4 shows a plot of the average processing time for HS and PBHS. A further comparison was done between sequential and parallel versions of both approaches. In this figure, HS-CPU, HS-GPU, PBHS-CPU and PBHS-GPU represent both approaches running in CPU and GPU. The time scale is shown in logarithmic scale due to its large span.

Figure 4 Comparison of processing time between implementations of HS and PBHS in CPU and GPU (see online version for colours)



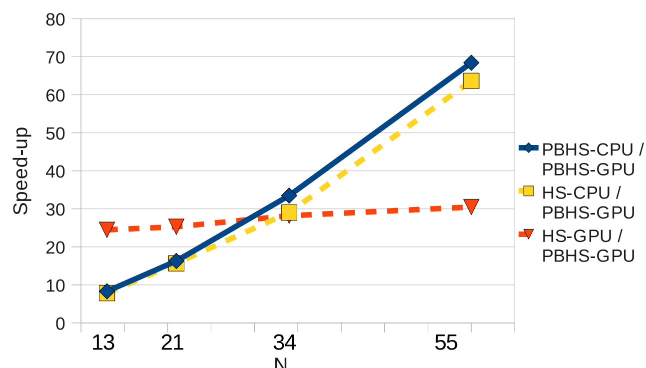
Overall, the processing time, for any approach, tends to increase sub-exponentially as the number of amino acids of the sequence increases. This fact strongly suggests the need for highly parallel approaches for dealing with the PSP.

For sequences with less than 34 amino acids, the HS-GPU presented an execution time above the HS-CPU. For $N = 34$ both were equivalent, and for $N = 55$ the HS-GPU presented a speed-up of 2x.

The PBHS-CPU version presented the worst performance (i.e., the highest processing time) for the sequences of 34 and 55 amino acids. Its processing time was slightly larger than HS-CPU, but very similar behaviour. This is due to the sequential nature of the selection procedure used in the improvisation process. The PBHS-GPU showed the best performance, mainly due the levels of parallelism in the process of improvisation and update, and the parallelisation of several function evaluations concurrently.

For a more comprehensive view of performance, Figure 5 presents the speed-ups of PBHS-GPU relative to HS-CPU, HS-GPU and PBHS-CPU, respectively.

Figure 5 Comparison of speed-ups between PBHS-GPU and the other approaches (see online version for colours)



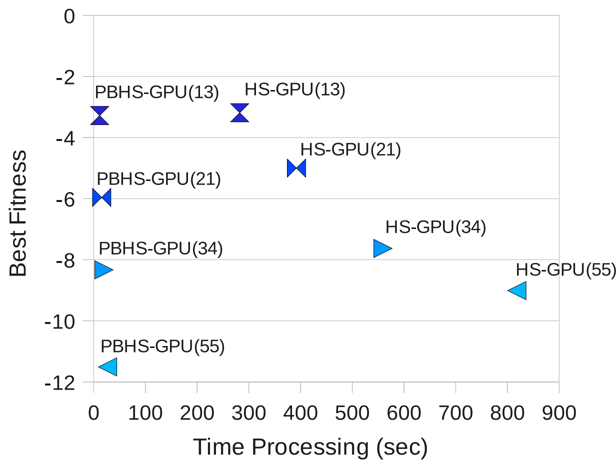
PBHS-GPU achieved significant speed-ups when compared to HS-GPU, ranging from 24.5x to 30.5x, depending on the length of the sequence. The smallest speed-up of PBHS-GPU was 7.8x, when compared with HS-CPU, for the sequence of 13 amino acids. The largest speed-up was 68.4x for the 55 amino acids-long sequence, when comparing PBHS-CPU and PBHS-GPU.

From Figure 5 we can observe that, as the length of sequences increases, the speed-up of GPU implementations regarding the CPU implementations increases exponentially. However, considering GPU implementations, the speed-up increases linearly. These facts strongly suggests not only the utility of GPU (instead of CPU) for scientific applications of growing complexity, but also, the gain of the parallel implementation over the original one.

A joint analysis of quality of solutions and performance was done by using the the concept of Pareto optimality (Deb, 2001). A plot is constructed in such a way to represent the behaviour of that two criterion, each of them to be minimised: the smaller the free energy, the better; and the smaller the processing time, the larger the performance. Each point in the plot represents a possible combination of parameters. In our case, the x axis represents the average processing time. The y axis is the best solution found ($BestFitness$). Each point of the plot (x_i, y_i) is classified as dominated, when there is at least another point (x_j, y_j) such that $(x_j < x_i) \bigwedge (y_j < y_i)$, or non-dominated, otherwise. In this analysis, only the non-dominated points are really interesting, since they represent the best possible trade-off between the two criterion. Notwithstanding, the Pareto plot also allows the user to find the most suitable working point for particular situations.

In Figure 6, it is possible to observe the differences between the non-dominated points. Each sequence is identified by different symbols and labels. The PBHS results are clearly identified as non-dominated having better processing times and quality of solutions than the other approaches.

Figure 6 Pareto plot comparing processing time and solution quality of all implementations (see online version for colours)



6 Conclusions and future work

This work presented the implementation of a Population-based Harmony Search algorithm running in GPU and in CPU. Experiments were done using the PSP problem, using the non-lattice 2D-AB model. We compare the proposed approach with the original (sequential) HS, taking into account not only the quality of solutions, but also, the performance (regarding processing time).

Results show that simply running an algorithm in GPU does not warrant a significant reduction of processing time, specially when dealing with small instances (that was evident in the case of HS-GPU). Still considering HS-GPU, a small gain in speed-up above 1x was achieved only with lengthiest sequence. This suggests that a sequential algorithm with small amount of data may be more efficiently run in CPU than in GPU.

On the other hand, as the implicit parallelism of an algorithm is exploited, significant speed-ups can be achieved when running in GPU, thanks to its structure that allows, for instance, 32 simultaneous function evaluations. Consequently, as the complexity increases (in this case, complexity is meant both, the amount of computation and the amount of data to be processed), the advantages of GPU processing are much more visible. This is evident when both implementations in GPU were compared with the equivalent in CPU, presenting an increment in the speed-up as the size of sequences increase.

For all test cases, the proposed population-based approach (PBHS) was significantly better, regarding quality of solutions, than the original HS. This was due to the capability of PBHS to maintain diversity longer than HS, thus avoiding premature convergence. The improvement in quality of solutions of PBHS over HS is evidenced in the average and best fitness for all the sequences.

Overall, this work showed that, by using a well-tuned parallelisation strategy, the GPU really can be useful for reducing the computational cost of populational metaheuristics, such as PBHS. For sure, this assertion could be extrapolated to other evolutionary computing paradigms, such as genetic algorithms, particle swarm optimisation and ant colony optimisation, for instance, where the evaluation of individuals can be done in parallel.

Future work will be towards the improvement of the PBHS, aiming at improving the quality of solutions, with the use of enhanced search mechanisms, and the processing performance, exploring more efficient parallelisation schemes in GPU. We believe that this combination is very promising for complex, large-scale scientific and engineering applications.

References

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K. and Walter, P. (2002) *Molecular Biology of the Cell*, Garland Science, New York, USA.

Almeida, C.P., Gonçalves, R.A. and Delgado, M.R.B.S. (2007) ‘A hybrid immune-based system for the protein folding problem’, *Lecture Notes in Computer Science*, Vol. 4446, pp.13–24.

Atkins, J. and Hart, W.E. (1999) ‘On the intractability of protein folding with a finite alphabet’, *Algorithmica*, Vol. 25, Nos. 2–3, pp.279–294.

Benítez, C.M.V. and Lopes, H.S. (2010) ‘Hierarchical parallel genetic algorithm applied to the three-dimensional HP side-chain protein folding problem’, *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, IEEE Computer Society, pp.2669–2676.

Berger, B. and Leighton, F.T. (1998) ‘Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete’, *Journal of Computational Biology*, Vol. 5, No. 1, pp.27–40.

Berman, H.M., Westbrook, J., Feng, Z., Gilliland, G. et al. (2000) ‘UniProt archive’, *Nucleic Acids Research*, Vol. 28, No. 1, pp.235–242.

Bitello, R. and Lopes, H.S. (2007) ‘A differential evolution approach for protein folding’, *Journal of Computer Science and Technology*, Vol. 22, No. 6, pp.904–908.

Blickle, T. (2000) ‘Tournament selection’, in T. Back, D.B. Fogel and Z. Michalewicz (Eds.): *Evolutionary Computation – Basic Algorithms and Operators*, pp.181–186, Institute of Physics Publishing, Bristol, UK.

Box, G.E., Hunter, W.G. and Hunter, J.S. (2005) *Statistics for Experimenters: Design, Innovation, and Discovery*, 2nd ed., J. Wiley & Sons, New York, NY, USA.

Bozejko, W., Smutnicki, C. and Uchronski, M. (2009) ‘Parallel calculating of the goal function in metaheuristics using GPU’, *Proceedings of the 9th International Conference on Computational Science*, Part I, pp.1014–1023, Springer-Verlag, Berlin, Heidelberg.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P. (2004) ‘Brook for GPUs: stream computing on graphics hardware’, *ACM Transactions on Graphics*, Vol. 23, No. 3, pp.777–786.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W. and Skadron, K. (2008) ‘A performance study of general-purpose applications on graphics processors using CUDA’, *International Journal of Parallel and Distributed Computing*, Vol. 68, No. 10, pp.1370–1380.

Cooper, G. (2000) *The Cell: A Molecular Approach*, Sinauer Associates, Sunderland, UK.

Cox, G.A., Mortimer-Jones, T.V., Taylor, R.P. and Johnston, R.L. (2004) 'Development and optimisation of a novel genetic algorithm for studying model protein folding', *Theoretical Chemistry Accounts*, Vol. 112, No. 3, pp.163–178.

Crescenzi, P., Goldman, D., Papadimitrou, C., Piccolboni, A. and Yannakakis, M. (1998) 'On the complexity of protein folding', *Journal of Computational Biology*, Vol. 5, No. 3, pp.423–446.

Cutello, V., Narzisi, G. and Nicosia, G. (2005) 'A class of Pareto archived evolution strategy algorithms using immune inspired operators for *ab-initio* protein structure prediction', *Applications on Evolutionary Computing*, Vol. 3449, pp.54-63, Springer-Verlag, Heidelberg, Germany.

Deb, K. (2001) *Multi-Objective Optimization using Evolutionary Algorithms*, John Wiley & Sons, Chichester, UK.

Dill, K.A., Bromberg, S., Yue, K., Fiebig, K.M. et al. (1985) 'Principles of protein folding – a perspective from simple exact models', *Protein Science*, Vol. 4, No. 4, pp.561–602.

Dill, K.A. (1999) 'Polymer principles and protein folding', *Protein Science*, Vol. 8, No. 6, pp.1166–1180.

Fesanghary, M., Mahdavi, M., Minary-Jolandan, M. and Alizadeh, Y. (2008) 'Hybridizing harmony search algorithm with sequential quadratic programming for engineering optimization problems', *Computer Methods in Applied Mechanics and Engineering*, Vol. 197, Nos. 33–40, pp.3080–3091.

Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y. and Volkov, V. (2008) 'Parallel computing experiences with CUDA', *IEEE Micro*, Vol. 28, No. 4, pp.13–27.

Geem, Z.W., Kim, J-H. and Loganathan, G.V. (2001) 'A new heuristic optimization algorithm: harmony search', *Simulation*, Vol. 76, No. 2, pp.60–68.

Geem, Z.W. (2009) *Music-Inspired Harmony Search Algorithm: Theory and Applications*, 1st ed., Springer, Heidelberg, Germany.

Geem, Z.W. (Ed.) (2010) 'State-of-the-art in the structure of harmony search algorithm', *Recent Advances in Harmony Search Algorithm*, Vol. 270, pp.1–10, Springer.

Hardin, C., Pogorelov, T.V. and Luthey-Schulten, Z. (2002) '*Ab-initio* protein structure prediction', *Current Opinion in Structural Biology*, Vol. 12, No. 2, pp.176–181.

Hsu, H.P., Mehra, V. and Grassberger, P. (2003) 'Structure optimization in an off-lattice protein model', *Physical Review E*, Vol. 68, No. 3, pp.id. 037703.

Hunter, L. (1993) *Artificial Intelligence and Molecular Biology*, 1st ed., AAAI Press, Boston, USA.

Irback, A., Peterson, C. and Potthast, F. (1997) 'Identification of amino acid sequences with good folding properties in an off-lattice model', *Physical Review E*, Vol. 55, No. 1, pp.860–867.

Jiang, T., Cui, Q., Shi, G. and Ma, S. (2003) 'Protein folding simulations of the hydrophobic-hydrophilic model by combining tabu search with genetic algorithms', *Journal of Chemical Physics*, Vol. 119, No. 8, pp.4592–4596.

Kalegari, D. and Lopes, H.S. (2010) 'A differential evolution approach for protein structure optimisation using a 2D off-lattice model', *International Journal of Bio-Inspired Computation*, Vol. 2, Nos. 3/4, pp.242–250.

Kirk, D. and Hwu, W. (2008) 'Applied parallel programming, Chapter 4 – CUDA memories', Draft.

Kirk, D. and Hwu, W. (2009a) 'Lectures 8: threading and memory hardware in G80', Draft.

Kirk, D. and Hwu, W. (2009b) 'Lectures 9: memory hardware in G80', Draft.

Komatitsch, D., Erlebacher, G., Goddeke, D. and Michea, D. (2010) 'High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster', *Journal of Computational Physics*, Vol. 229, No. 20, pp.7672–7714.

Lee, K.S. and Geem, Z.W. (2004) 'A new structural optimization method based on the harmony search algorithm', *Computers & Structures*, Vol. 82, Nos. 9–10, pp.781–798.

Lodish, H., Berk, A., Matsudaira, P., Kaiser, C.A., Krieger, M., Scott, M.P., Zipursky, L. and Darnell, J. (2000) *Molecular Cell Biology*, 4th ed., Freeman, New York, NY, USA.

Lopes, H.S. (2008) 'Evolutionary algorithms for the protein folding problem: a review and current trends', in Smolinski, T.G., Milanova, M.M. and Hassanien, A-E. (Eds.): *Computational Intelligence in Biomedicine and Bioinformatics*, Vol. I, pp.297–315, Springer-Verlag, Heidelberg, Germany.

Mahdavi, M., Fesanghary, M. and Damangir, E. (2007) 'An improved harmony search algorithm for solving optimization problems', *Applied Mathematics and Computation*, Vol. 188, No. 2, pp.1567–1579.

Mohanty, S.P. (2009) 'GPU-CPU multi-core for real-time signal processing', *Proc. of the 27th IEEE International Conference on Consumer Electronics*, pp.55–56, IEEE Computer Society, Los Alamitos, USA.

Moorkamp, M., Jegen, M., Roberts, A. and Hobbs, R. (2010) 'Massively parallel forward modeling of scalar and tensor gravimetry data', *Computers & Geosciences*, Vol. 36, No. 5, pp.680–686.

NVIDIA (2007) *CUDA for GPU Computing*, NVIDIA Corporation, February.

NVIDIA (2008) *CUDA Programming Model Overview*, NVIDIA CUDA Team.

NVIDIA (2009) *NVIDIA Compute PTX: Parallel Thread Execution*, ISA Version 1.4, NVIDIA CUDA Team, June.

NVIDIA (2010) *CUDA Programming Guide Version 3.0*, NVIDIA CUDA Team, February.

Ostrovsky, B., Crooks, G., Smith, M.A. and Bar-Yam, Y. (2001) 'Cellular automata for polymer simulation with application to polymer melts and polymer collapse including implications for protein folding', *Parallel Computing*, Vol. 27, No. 5, pp.613–641.

Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. and Purcell, T.J. (2007) 'A survey of general-purpose computation on graphics hardware', *Computer Graphics Forum*, Vol. 26, No. 1, pp.80–113.

Parhami, B. (2002) *Introduction to Parallel Processing: algorithms and structures*, Kluwer Academic, New York.

Parpinelli, R.S. and Lopes, H.S. (2011) 'New inspirations in swarm intelligence: a survey', *International Journal of Bio-Inspired Computation*, Vol. 3, No. 1, pp.1–16.

Saka, M.P. (2007) 'Optimum geometry design of geodesic domes using harmony search algorithm', *Advances in Structural Engineering*, Vol. 10, No. 6, pp.595–606.

Shams, R., Sadeghi, P., Kennedy, R. and Hartley, R. (2010) 'Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images', *Computer Methods and Programs in Biomedicine*, Vol. 99, No. 2, pp.133–146.

Shmygelska, A. and Hoos, H.H. (2005) 'An ant colony optimisation algorithm for the 2D and 3D hydrophobic polar protein folding problem', *BMC Bioinformatics*, Vol. 6, No. 30, pp.1–22.

Stillinger, F.H. and Head-Gordon, T. (1995) 'Collective aspects of protein folding illustrated by a toy model', *Physical Review E*, Vol. 52, No. 3, pp.2872–2877.

Stillinger, F.H., Head-Gordon, T. and Hirshfeld, C. (1993) 'Toy model for protein folding', *Physical Review E*, Vol. 48, No. 2, pp.1469–1477.

Sunarso, A., Tsuji, T. and Chono, S. (2010) 'GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows', *Journal of Computational Physics*, Vol. 229, No. 15, pp.5486–5497.

Tan, Y. and Zhou, Y. (2010) 'Parallel particle swarm optimization algorithm based on graphic processing units', in Hiot, L.M., Ong, Y.S., Panigrahi, B.K., Shi, Y. and Lim, M-H. (Eds.): *Handbook of Swarm Intelligence*, Vol. 8 of *Adaptation, Learning, and Optimization*, pp.133–154, Springer, Heidelberg, Germany.

Tantar, A.A., Melab, N., Talbi, E.G., Parent, B. and Horvath, D. (2007) 'A parallel hybrid genetic algorithm for protein structure prediction on the computational grid', *Future Generation Computer Systems*, Vol. 23, No. 3, pp.398–409.

Torcini, A., Livi, R. and Politi, A. (2001) 'A dynamical approach to protein folding', *Journal of Biological Physics*, Vol. 27, Nos. 2–3, pp.181–203.

Yanikoglu, B. and Erman, B. (2002) 'Minimum energy configurations of the 2-dimensional HP-model of proteins by self-organizing networks', *Journal of Computational Biology*, Vol. 9, No. 4, pp.613–620.

Yu, Q., Chen, C. and Pan, Z. (2005) 'Parallel genetic algorithms on programmable graphics hardware', *Lecture Notes in Computer Science*, Vol. 3612, pp.1051–1059.

Zhang, X. and Cheng, W. (2008) 'An improved tabu search algorithm for 3D protein folding problem', *Lecture Notes in Computer Science*, Vol. 5351, pp.1104–1109.

Notes

1 For more information see the HS repository: http://www.hydroteq.com.

2 See http://www.ebi.ac.uk/uniprot/ for updated information.

3 See http://http://www.pdb.org for updated information.

4 Available at http://www.nvidia.com/object/cuda_home_new.html.