UNIVERSITÀ DEGLI STUDI DI PARMA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Dottorato di Ricerca in Tecnologie dell'Informazione

XXVI Ciclo

Youssef S. G. Nashed

PARALLEL BIO-INSPIRED METHODS FOR MODEL OPTIMIZATION AND PATTERN RECOGNITION

DISSERTAZIONE PRESENTATA PER IL CONSEGUIMENTO

DEL TITOLO DI DOTTORE DI RICERCA

GENNAIO 2014

UNIVERSITÀ DEGLI STUDI DI PARMA

Dottorato di Ricerca in Tecnologie dell'Informazione

XXVI Ciclo

PARALLEL BIO-INSPIRED METHODS FOR MODEL OPTIMIZATION AND PATTERN RECOGNITION

Coordinatore:
Chiar.mo Prof. Marco Locatelli

Tutor:
Chiar.mo Prof. Stefano Cagnoni

Dottorando: *Youssef S. G. Nashed*

Gennaio 2014

To my beloved wife, Sandra

Contents

Acknowledgements **1**

Abstract **3**

1 Introduction **5**

2 Background **9**

2.1 Metaheuristics 9

Particle Swarm Optimization 10

Differential Evolution 12

Scatter Search 15

2.2 The Neocortex 19

Memory Prediction Framework 21

2.3 General-Purpose GPU Programming 22

NVIDIA GPU Architecture 23

CUDA Programming Model 23

3 Parallel Metaheuristics **27**

3.1 CUDA Particle Swarm Optimization 28

CUDA Asynchronous PSO 29
Implementation 30
3.2 CUDA Differential Evolution 34
Implementation 34
3.3 CUDA Scatter Search 35
Implementation 36
3.4 libCudaOptimize 37
Implementation 37
Usage 38
3.5 Testing and Results 40
Speedup Results 40
Benchmark Functions 43
Real-World Application 48
3.6 Final Remarks 56

4 Hierarchical Quilted Self Organizing Maps 59

4.1 Self-Organizing Maps 61
Recurrent Self Organizing Maps 62
Parameter-less Self Organizing Maps 62
4.2 Multi-modal Pattern Recognition with HQSOM 65
Implementation 66
Testing and Results 68
4.3 Final Remarks 76

5 Automatic Configuration of the HQSOM **77**

5.1 Parameter Tuning 77

5.2 HQSOM Tuning via Real Parameter Optimization 78

Model Formulation 79

Fitness Function 80

Testing and Results 82

5.3 Final Remarks 88

6 Further Work **91**

7 Summary and Conclusions **95**

Bibliography **99**

List of Figures

2.1 The Neocortex lobes 19

2.2 Face Recognition in the Visual Cortex 20

2.3 HTM Structure 21

2.4 CUDA Grids 24

2.5 CUDA Kernel Execution 25

2.6 CUDA Memory Hierarchy 26

3.1 PSO Topologies 30

3.2 Block Diagram of the Synchronous CUDA PSO algorithm 31

3.3 Synchronous CUDA-PSO VS Asynchronous CUDA-PSO 33

3.4 Block Diagram of the CUDA DE algorithm 35

3.5 Block diagram of the Scatter Search Algorithm. 36

3.6 libCudaOptimize UML diagram 38

3.7 CUDA-PSO test results 42

3.8 Benchmark Function Plots 46

3.9 Body Pose Estimation Per-joint Plots 53

3.10 Body Pose Estimation Per-frame Plots 53

3.11 Body Pose Estimation PSO and DE comparison 54
3.12 Body Pose Estimation Example Results 55

4.1 HQSOM structure 60
4.2 Simulated moving arrow sequences 69
4.3 Arrows HQSOM weights after training 71
4.4 ChaLearn training samples 72
4.5 ChaLearn HQSOM weights after training 75

5.1 Box plots of DE,PSO,`irace`, and manually tuned parameters 86
5.2 DE VS PSO parameter estimation 87

List of Tables

3.1 Pseudo-code for the sequential versions of PSO 32

3.2 Automatically-tuned parameters for libCudaOptimize 43

3.3 Benchmark Function Definitions 44

3.4 Results on the 20 benchmark functions. 47

3.5 Parameters used for human body pose estimation 50

3.6 Body Pose Estimation distance results 51

3.7 Body Pose Estimation fitness results 52

3.8 Body Pose Estimation fitness results on all the videos 52

4.1 Arrows HQSOM Parameters 69

4.2 ChaLearn HQSOM Parameters 73

4.3 ChaLearn HQSOM Results 74

5.1 Parameters of DE/PSO tuners. 83

5.2 HQSOM parameters to be automatically estimated 84

5.3 ChaLearn HQSOM accuracy results with auto-tuned parameters . . . 85

5.4 ChaLearn HQSOM Levenshtein distances with auto-tuned parameters 85

5.5 Parameter sets found by both PSO, and `irace`. 86

Acknowledgements

The work presented in this thesis was carried out in the Intelligent Bio-Inspired System (IBIS) Laboratory of the Department of Information Engineering of the University of Parma, Italy. It would not have been possible to write this doctoral thesis without the help and support of the kind people around me, to only some of whom it is possible to give particular mention here.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Stefano Cagnoni, for his excellent guidance, caring, patience, and providing me with an excellent atmosphere for doing research. He kept an open mind to new ideas, while providing valuable scientific and technical context to them. Also on the administrative and personal issues, he was always flexible and willing to lend a helping hand when needed. Grazie professore!

Secondly, I have to specially thank my colleagues and friends in the IBIS lab, Pablo Mesejo and Roberto Ugolotti, without whom many of the projects mentioned in this thesis would not have seen the light. They have provided me with many ideas and clues on how to tackle my technical, scientific, and personal problems. We have spent a lot of time together on writing papers, debugging code, attending conferences, supervising students, making jokes, drinking countless cups of coffee, and consuming a copious amount of pasta. They have made time a lot more enjoyable and provided input and technical aid to many aspects of my projects.

Besides the people at the University of Parma, I would like to extend my sincere thanks to Luca Mussi, from Henesis s.r.l. He took time from his busy schedule to collaborate with me on my first publication, and although most of the communication

was done remotely, I thank him for his patience in introducing me to CUDA, which all of my work depends on. He was also the one that pointed me towards the Hierarchical Quilted Self Organizing Maps model during one of his rare visits to the University.

My doctorate degree was funded by the European Comission (Marie Curie ITN MIBISOC, FP7 PEOPLE-ITN-2008, GA n. 238819). This time involved many workshops, activities, and two secondments abroad. I would like to thank the MIBISOC network management team, more specifically Oscar Cordón and Carmen Peña, for offering me this opportunity and arranging all the activities and paperwork. Also, a warm thank you for all my fellow MIBISOC early-stage researchers, it was a pleasure having all those meetings and discussions together.

Last but not the least, I am grateful to my friends and family back home for their continuous love and encouragement. To my parents for raising me up with the love of reading and science, and their prayers for my studies abroad. I am most grateful to my wife, Sandra, who supported me throughout the good and bad times, specially during the final stages of writing this thesis. Thank you.

Los Angeles, September 6, 2013

Youssef S. G. Nashed

Abstract

Nature based computational models are usually inherently parallel. The collaborative intelligence in those models emerges from the simultaneous instruction processing by simple independent units (neurons, ants, swarm members, etc...). This dissertation investigates the benefits of such parallel models in terms of efficiency and accuracy. First, the viability of a parallel implementation of bio-inspired metaheuristics for function optimization on consumer-level graphic cards is studied in detail. Then, in an effort to expose those parallel methods to the research community, the metaheuristic implementations were abstracted and grouped in an open source parameter/function optimization library *libCudaOptimize*. The library was verified against a well known benchmark for mathematical function minimization, and showed significant gains in both execution time and minimization accuracy. Crossing more into the application side, a parallel model of the human neocortex was developed. This model is able to detect, classify, and predict patterns in time-series data in an unsupervised way. Finally, libCudaOptimize was used to find the best parameters for this neocortex model, adapting it to gesture recognition within publicly available datasets.

Chapter 1

Introduction

Why are bio-inspired methods good models for intelligence? The answer to this question lies in the difference between bio-inspired computation and classical Artificial Intelligence (AI). In traditional AI, the programmer has all the knowledge, and encodes the intelligent behavior within the system from above. On the other hand, bio-inspired methods follow a bottom-up approach. In most cases of bio-inspired models, they consist of a set of individuals/organisms, each applying a simple set of rules, for a number of iterations or generations. A complex behavior arises from the collective basic individual's actions, accumulated after rule application cycles. Such a model is in accordance with the evolutionary approach to learning, where the simple rules are selection, combination/reproduction, and mutation, that through millions of years resulted in extremely complex structures and creatures. In less technical terms, the most obvious reason to use nature-based methods is that we know, from everything around us, that they actually work.

Bio-inspired methods include, but are not limited to, Genetic Algorithms (GA), Artificial Neural Networks (ANN), Ant Colony Optimization (ACO), Artificial Life (ALife), and Swarm Intelligence (SI) approaches. As specified above, using any of these methods entails the simulation of a group of instruction processing units, running for several iterations, which in turn limits the applicability of these approaches, because of the required computational load. This reason, along with the fact that they

are decentralized methods, makes bio-inspired algorithms excellent candidates for parallelization. As far as we are concerned in this study, we present parallel instances of the SI algorithms, which are used for real-valued parameter estimation and function optimization. Another aspect that is of great interest to the authors is the real-time detection of patterns or anomalies in time series data (i.e. videos, range sensor data, audio, etc...). Here, this is achieved through a new parallel implementation of a neural model that is called Hierarchical Quilted Self Organizing Maps (HQSOM). One of the main contributions of this research is employing function optimization techniques to evolve variants of the HQSOM that can adapt to the pattern classification task, regardless of the dataset nature.

The term function or mathematical optimization may sound too technical to the unfamiliar reader. However, optimization is rather widely used in real life applications. It is defined as trying to find a set of values to variables, or parameters, of a function that give the maximum or minimum output ("objective function" or, when evolutionary computing algorithms are considered, "fitness function"). For example, let us say you want to buy a new house, and for simplicity purposes, let us also say you are considering two factors only when searching for your new home: house area, and neighborhood. Surely, you would want to pay as little as possible. In this case, we can consider this decision as a two-dimensional function optimization problem, a two-dimensional minimization to be more precise. The function parameters/dimensions are area, and neighborhood, while its output/fitness is price. Optimization methods are designed to deal with such problems, to find good values for the function parameters that give the optimal (or near optimal) fitness. Optimization is used in many applications, from different fields, ranging from engineering and aerospace design optimization, to economics and operations research.

Real-time pattern detection and understanding in multimodal environments is becoming paramount to applications from different fields. Automatic surveillance systems and assisted-living homes will benefit from research done in areas such as human activity classification and object detection, usually involving temporal sequences from video cameras or other wearable sensors. Gesture and voice recognition can be useful

for human robot interaction systems, sign language interfaces, and even gaming. Fortunately, the video games industry currently fuels a huge market, pushing innovation in the design and manufacturing process of graphic cards and intuitive gaming console controllers. The project at hand is more specifically interested in nVIDIA's CUDA parallel programming framework, in addition to the increasingly popular Microsoft Kinect^{TM}sensor.

The rest of this work is organized into four sections. Chapter 2 provides an overview of the literature, and terms used throughout the dissertation. It presents the notion of metaheuristics, explaining in detail the three methods implemented, in addition to the Memory Prediction Framework (MPF), on which the HQSOM model is envisioned. The parallel programming framework that supports all the methods implemented is also presented in this chapter. In Chapter 3, we expand on the concept of parallel metaheuristics, providing implementation details on the actual algorithms, and the open source library used in this project. Testing on a benchmark of well known mathematical functions, results are shown in terms of convergence to the function minimum, speedup gain compared to a sequential method, an assessment of the parallelism potential of each metaheuristic, and finally, on a case study of a real-world application. The HQSOM model for pattern detection and classification is presented in Chapter 4. The basic building model of the HQSOM is the Self-Organizing Map (SOM) algorithm. There are several variants of this algorithm, to adapt it to clustering different kinds of data. We will explain the SOM algorithm, the modifications made to it, and the datasets used to verify correctness of the model. Chapter 5 introduces the novel technique of finding a good parameter set for the HQSOM through optimization by metaheuristics, effectively decoupling the classifier from the modality and properties of the dataset. Finally, Chapters 6 and 7 include some final remarks and a discussion about possible future developments.

Chapter 2

Background

In this chapter we review the key concepts required to understand the research project at hand. The motivation behind using metaheuristics for continuous optimization is provided below, giving more attention to the three bio-inspired optimization techniques implemented to execute in parallel. Moving on to a seemingly different subject, the chapter continues with the biological and theoretical basis of the HQSOM model, using a similar approach to explain the requirements for such models. Lastly, the final section describes the programming environment CUDA by nVIDIA, within which we have developed and implemented our methods, to help the readers' understanding of the implementation choices which will be described further on.

Although the following sections may appear unrelated to the reader, upcoming chapters will provide the common ground where those different subfields of research come together in a single application.

2.1 Metaheuristics

A heuristic search method is the one that uses specific knowledge about a problem to find the solution. As for metaheuristics, they are a family of algorithms that are mainly used as global search methods for function optimization. Metaheuristics do not use

problem-specific knowledge, but they make assumptions about the problem class and good solution locations (fitness landscape). Therefore, it is virtually impossible to find a metaheuristic that can solve all kinds of optimization problems [1]. Metaheuristics are usually stochastic methods, starting with randomly chosen feasible solutions, then selecting the best one(s) as a guide for future algorithm iterations. This random component of metaheuristics make them non-deterministic methods, which in turn does not guarantee optimality. However, they are less computationally complex than exact methods. Moreover, in practice, they are known to find near optimal solutions in very few iterations (fast convergence).

Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a bio-inspired optimization algorithm introduced by Kennedy and Eberhart [2]. It is based on the simulation of the social behavior of bird flocks. In the last fifteen years PSO has been applied to a very large variety of problems [3] and many variants of the original algorithm have been proposed [4].

During the execution of PSO, a set of particles moves within a function domain searching for the optimum of the function (best fitness value). The motion of the i^{th} ($i = 1, N_p$) particle can be described by the following two simple equations which regulate the particle's position and velocity:

$$\begin{aligned} v_i(t) &= w \cdot v_i(t-1) \\ &+ c_1 \cdot rand(0,1) \cdot (BP_i - P_i(t-1)) \\ &+ c_2 \cdot rand(0,1) \cdot (BGP - P_i(t-1)) \\ P_i(t) &= P_i(t-1) + v_i(t) \end{aligned}$$

where $v_i(t)$ and $P_i(t)$ are respectively the velocity and position of the particle in the present iteration, c_1, c_2 and w (inertia factor) are positive constants, $rand(l, u)$ returns random values uniformly distributed in $[l, u]$, BP_i is the best-fitness position visited so far by the particle. In the basic algorithm "global-best PSO", BGP is the best-fitness

position visited so far by any particle of the swarm. In several variants, termed "local-best PSO", the swarm is subdivided into particle neighborhoods which can assume different topologies. In that case, BGP becomes BGP_i and represents the best-fitness position visited so far by any particle in the i^{th} particle's neighborhood. Among the possible neighborhoods, the ring-shaped one is particularly interesting for its simple implementation, as well as for the role it may play in optimizing the efficiency of PSO parallelization [5] and even, sometimes, for the improvement of convergence speed [6] it may bring.

Formally, let $f(P)$ be the fitness function under optimization (to be minimized), where $P = [p_1, p_2, \cdots, p_D]$ is a candidate solution in the form of a real valued vector of size D = the problem dimension, and l_i and u_i are the lower and upper bounds of the i^{th} dimension, respectively.

The basic PSO algorithm is then defined as:

for each patricle P_i **do**
  for each dimension $d \leftarrow 1, D$ **do**
    $P_i[d] \leftarrow rand(l_d, u_d)$
    $V_i[d] \leftarrow rand(-|u_d - l_d|, |u_d - l_d|)$
    $BP_i[d] \leftarrow P_i[d]$
  end for
  if $f(P_i) < f(BGP)$ **then**
    $BGP \leftarrow P_i$
  end if
end for
while termination criteria is not met **do**
  for each patricle P_i **do**
    for each dimension $d \leftarrow 1, D$ **do**
      $r_p \leftarrow rand(0, 1)$
      $r_g \leftarrow rand(0, 1)$
      $V_i[d] \leftarrow wV_i[d] + c_1 r_p BP_i[d] + c_2 r_g BGP_i[d]$
      $P_i[d] \leftarrow P_i[d] + V_i[d]$

```
        end for
        if f(P_i) < f(BP_i) then
            BP_i ← P_i
            if f(P_i) < f(BGP) then
                BGP ← P_i
            end if
        end if
    end for
end while
```

At the end of the PSO algorithm BGP will hold the best found solution.

Differential Evolution

Differential Evolution (DE), first introduced by Storn and Price [7], has recently been one of the most successful Evolutionary Algorithms for global continuous optimization, especially when the function to be optimized is multimodal and non-separable [8]. Unlike traditional EAs, DE perturbs the individuals of the current generation by the scaled differences of other randomly-selected and distinct individuals. Therefore, no separate probability distribution has to be used for generating the offspring [9]. This way, in the first iterations the population members are widely scattered in the search space and possess great exploration ability. During optimization, the individuals tend to concentrate in the regions of the search space with better values, so the search automatically focuses onto the most promising areas [10].

In DE, new individuals that will be part of the next generation are created by combining individuals that are already members of the current population. Every individual acts as a parent vector and, for each of them, a donor vector is created. In the basic version of DE, the donor vector for the i^{th} parent (X_i) is generated by combining three random and distinct individuals X_{r1}, X_{r2} and X_{r3}. The donor vector V_i is calculated by what is called mutation of difference vectors as follows:

$$V_i = X_{r1} + F \cdot (X_{r2} - X_{r3})$$

where F (scale factor) is a parameter that strongly influences DE's performances and typically lies in the interval $[0.4,\ 1]$. Recently, several mutation strategies have been applied to DE, experimenting with different base vectors and different numbers of vectors for perturbations. For example, the original method explained above is called DE/rand/1, which means that the first element of the donor vector equation X_{r1} is randomly chosen and only one difference vector (in our case $X_{r2} - X_{r3}$) is added. After mutation, every parent-donor pair generates a child (U_i), called trial vector, by means of a crossover operation.

$$U_{i,j} = \begin{cases} V_{i,j} & \text{if } (rand(0,1) \leq C_r \text{ or } j = j_{rand}) \\ X_{i,j} & \text{otherwise} \end{cases}$$

As described in the above equation, the j^{th} component/dimension of the i^{th} donor vector is obtained by means of uniform (binomial) crossover, where C_r is the crossover rate, and j_{rand} is a randomly selected dimension. The newly-generated individual U_i is evaluated by comparing its fitness to its parent's fitness. The best survives and will be part of the next generation.

DE shares some features with swarm intelligence techniques, mainly related with the interaction among particles and the selection scheme. In particular, both DE and PSO are stochastic, population based, real-valued algorithms, and designed for challenging continuous optimization problems (non-differentiable, nonlinear and/or multimodal functions) using few control parameters. DE can also be considered as an Evolutionary Algorithm (EA), but differs from traditional EA algorithms in the aspect of generating new vectors by adding the weighted difference vector between two population members to a third member.

The basic DE algorithm, with random mutation and binomial crossover, can then be described as follows:

for each candidate solution X_i, $i \leftarrow 1, N$ **do**
  for each dimension $d \leftarrow 1, D$ **do**
    $X_i[d] \leftarrow rand(l_d, u_d)$
  end for
end for
while termination criteria is not met **do**
  for each candidate solution X_i, $i \leftarrow 1, N$ **do**
    $r1 \leftarrow r2 \leftarrow r3 \leftarrow i$
    while $r1$, $r2$, $r3$ and i are not mutually exclusive integers **do**
      $r1 \leftarrow rand(1, N)$
      $r2 \leftarrow rand(1, N)$
      $r3 \leftarrow rand(1, N)$
    end while
    $j \leftarrow rand(1, D)$
    for each dimension $d \leftarrow 1, D$ **do**
      if $d \equiv j \vee rand(0, 1) \leq C_r$ **then**
        $U_i[d] \leftarrow X_{r1}[d] + F(X_{r2}[d] - X_{r3}[d])$
      else
        $U_i[d] \leftarrow X_i[d]$
      end if
    end for
    if $f(U_i) < f(X_i)$ **then**
      $X_i \leftarrow U_i$
    end if
  end for
end while

The best candidate solution of the final generation is the best overall found solution.

Scatter Search

Scatter Search (SS), originally proposed by Glover [11], is based on a systematic combination between solutions (instead of a randomized one, as usually happens in EAs) taken from a considerably reduced evolved pool of solutions named the reference set (usually between five and ten times lower than typical EA population sizes). SS is composed of 5 structural "blocks" or methods:

1. Diversification Generation: a population of solutions P is built with a certain degree of quality and diversity. The reference set R is then drawn from P, and it is composed of the $|R_1|$ solutions with best fitness and the $|R_2|$ solutions with the maximum euclidean distance to the reference set; the evolution process works only over R;

2. Improvement: to obtain quality solutions, an improvement method is applied to original solutions and/or combined solutions (usually a "local search");

3. Reference Set Update: once a new solution is obtained (applying the combination method) it replaces the worst solution in R only if it improves the quality of the reference set (in terms of fitness and/or diversity);

4. Solution Combination: in most of the problems a specific solution combination method is needed, and it can be selectively applied and/or using random elements. In many cases an existing GA crossover operator can be employed;

5. Subset Generation: the procedure generates subsets from R, in a deterministic way, to which the combination method is applied. These combinations can be made considering pairs, triplets,....

Since SS is only a template for constructing many variants of the algorithm, the procedure for implementing it is not composed of concrete mathematical steps, rather it consists of guidelines on how to use its building blocks. The basic SS algorithm was outlined in [12] as follows:

1: Start with $P = \emptyset$. Use the diversification generation method to construct a solution and apply the improvement method. Let x be the resulting solution. If $x \notin P$ then add x to P (i.e. $P = P \cup x$), otherwise, discard x.
2: Repeat step 1 until $|P| = N$
3: Use the reference set update method to build $RefSet = \{x_1, x_2, \cdots, x_b\}$ with best b solutions in P. Order the solutions in $RefSet$ according to their fitness, such that x_1 is the best solution, and x_b the worst.
4: $NewSolutions \leftarrow TRUE$
5: **while** $NewSolutions$ **do**
6: Generate $NewSubsets$ with the subset generation method.
7: $NewSolutions \leftarrow FALSE$
8: **while** $NewSubsets \neq \emptyset$ **do**
9: Select the next subset s in $NewSubsets$
10: Apply the $solutioncombination$ method to s to obtain the trial solutions.
11: Apply the improvement method to the trial solutions.
12: Apply the reference set update method.
13: **if** $RefSet$ has changed **then**
14: $NewSolutions \leftarrow TRUE$
15: **end if**
16: Delete s from $NewSubsets$
17: **end while**
18: **end while**

Solis&Wets local search

Here we use Solis&Wets local search [13] as the improvement method of Scatter Search. Solis&Wets method is a randomized hill-climber with adaptive step size. Each step starts at a point x. A perturbation dif is randomly chosen from a Gaussian distribution with standard deviation ρ, for each problem dimension. If either $x + dif$ or $x - dif$ has a better fitness than x, a move to the best point is performed and a success is recorded, otherwise the position does not change and a failure is recorded. After N^+

consecutive successes ρ is increased, for getting faster to the local optima, while after N^- failures in a row, ρ is consequently decreased.

A single run of the Solis&Wets algorithm for a candidate solution x is described below:

function SOLISWETS(x, D, $bias$, ρ)
  $numEval \leftarrow 0$
  $numSuccess \leftarrow 0$
  $numFailed \leftarrow 0$
  while $numEval < maxEval$ **do**
    for $i \leftarrow 1, D$ **do**
      $dif[i] \leftarrow randGaussian(0, \rho)$
    end for
    $xp \leftarrow x + bias + dif$
    if $f(xp) < f(x)$ **then**
      $x \leftarrow xp$
      $bias \leftarrow 0.2 \times bias + 0.4 \times (dif + bias)$
      $numSucess \leftarrow numSuccess + 1$
      $numFailed \leftarrow 0$
    else
      $xp \leftarrow x - bias - dif$
      if $f(xp) < f(x)$ **then**
        $x \leftarrow xp$
        $bias \leftarrow bias - 0.4 \times (dif + bias)$
        $numSucess \leftarrow numSuccess + 1$
        $numFailed \leftarrow 0$
      else
        $numFailed \leftarrow numFailed + 1$
        $numSucess \leftarrow 0$
      end if
    end if
    if $numSucess > N^+$ **then**

$\rho \leftarrow 2\rho$

$numSucess \leftarrow 0$

else if $numFailed > N^-$ **then**

$\rho \leftarrow \rho/2$

$numFailed \leftarrow 0$

end if

$numEval \leftarrow numEval + 1$

end while

end function

2.2 The Neocortex

The cerebral cortex is the folded outer layer of the human and mammalian brains. Anatomically, it is composed of a thin layer (2 to 4 millimeters in thickness) of neural tissue, and covers the cerebrum, which is divided into two cortices, the left and right cerebral hemispheres. It is usually referred to as *gray matter*, because of the neuronal cell bodies and blood capillaries that run through it, making it darker than the underlying *white matter* areas, that is the complex network of neuronal axon bundles, or nerve cell endings, that connect parts of the cerebral cortex to each other, and other parts of the central nervous system.

The neocortex, also called the isocortex and neopallium, is the newest part of the cerebral cortex to evolve (hence the Latin prefix *neo*). The ratio of the size of the neocortex to the total size of the brain is thought to correlate to the intelligence of a species. In humans, the neocortex is 90% of the cerebral cortex. It is involved in higher brain functions, such as perception of sensory information, memory, spatial reasoning, language, and conscious thought. The neocortex is divided into frontal, parietal, occipital, and temporal lobes, each performing a different function, see Figure 2.1.

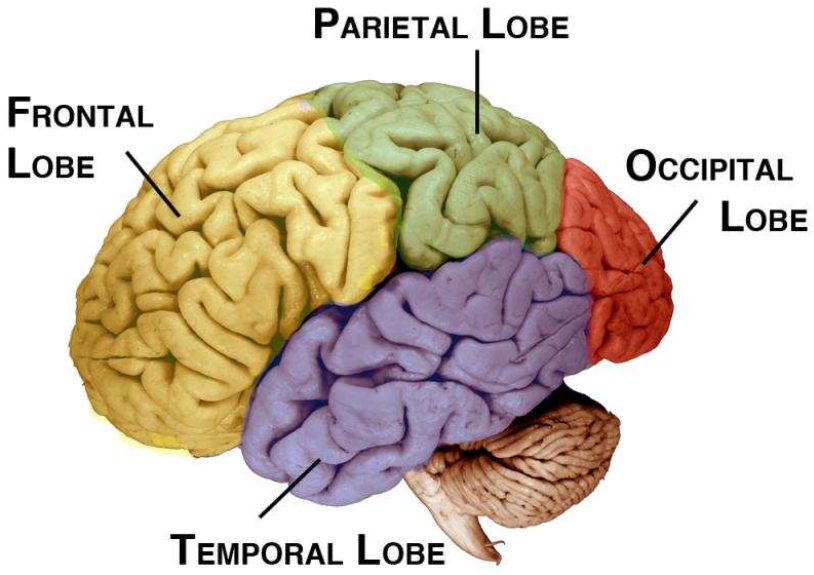


Figure 2.1: The Neocortex lobes

The neocortex is made up of layers, interconnected with feedforward and feedback connections. Lower-level layers detect simple features, passing them to higher levels that build associations of those features forming invariant abstract representations of a concept. For a more concrete example, let us consider the processing of visual information. In the primary visual cortex (part of the occipital lobe), the lowest level of neurons, known as V1, respond to low-level visual features, such as horizontal and vertical lines. Information from V1 is passed to higher levels (V2, V4, and V5), where some levels are more linked to motion, while others are responsible for storing long-term memory of object representations; this process is illustrated in Figure 2.2 for the face recognition task. On the other hand, feedback connections from higher levels to lower levels provide predictions for the currently sensed features based on previous experience/memory.

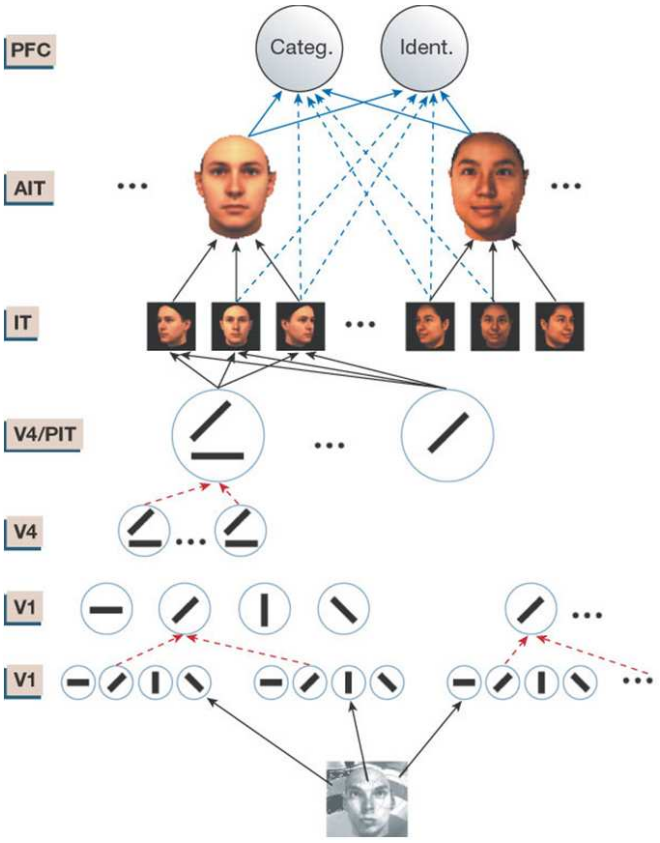


Figure 2.2: Face Recognition in the Visual Cortex [14]

Recent breakthroughs in the fields of neuroscience and functional magnetic resonance imaging have drawn attention to the role of the retrosplenial cortex (part of the temporal lobe) in recognizing the familiarity of a person, irrespective of the stimulus modality [15]. In other terms, during the task of distinguishing people known to a subject, the flow of information through the neocortex appear to be similar, whether the subject is presented with either faces or voices. Expanding on these findings, Jeff Hawkins proposed a computational model of the neocortex, the Memory-Prediction Framework (MPF), for identifying, clustering, and predicting patterns in any modality of temporal signals [16], (i.e. videos, audio, stock market data, etc...).

Memory Prediction Framework

The MPF is inspired by the structure explained in the previous section, where the basic unit comprising each level should perform both spatial and temporal pooling or clustering. Only the first level deals with the sensory information from the input signal, effectively decoupling the model from the modality of the signal. On the practical side, George and Hawkins describe their implementation of the MPF, the HTM [17], which is a Bayesian network with layers arranged in a tree-shaped hierarchy (Figure 2.3), based on the spatial correlations of the input data.

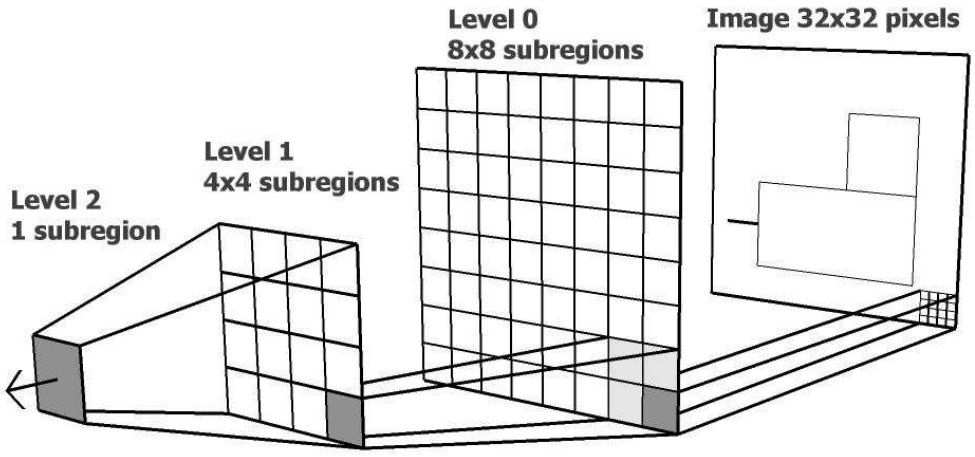


Figure 2.3: HTM Structure [18]

As shown in Figure 2.3, the lower-level layers of the HTM correspond to regions in the neocortex (V1 in the visual cortex) while, at the top, HTM has one output node, or neuron, that may play a similar role to the task of the Hippocampus in the mammalian brain. Although the time aspect is crucial in the MPF model theory, the HTM was only tested for stationary binary object recognition. Also, HTM suffers from the lack of feedback connections from higher to lower levels of its networks, which contradicts with the biological and theoretical model of the neocortex, and deprive the HTM from pattern-prediction capabilities. Moreover, the Bayesian basis of the HTM complicates dissecting and debugging the model during the training phase. In this work, we will try to address these shortcomings or provide future solutions for them.

2.3 General-Purpose GPU Programming

Modern graphics hardware has gained an important role in the area of parallel computing. Graphic cards have been used in 3D graphics applications and gaming but, recently, they have also been more and more frequently used to accelerate numeric computation, in what is usually called general-purpose GPU (GPGPU) programming [19]. The main advantage of using GPUs lies in their structure: while standard CPUs usually contain a handful of complex computational cores, memory registers and large cache memory, GPUs contain up to several hundreds of cores grouped into so-called multiprocessors, organized such that each ALU of a multiprocessor executes the same operations on different data, stored in registers or device memory. In contrast with standard CPUs, which can reschedule operations (out-of-order execution), current GPUs are an example of an in-order architecture, but this drawback can be overcome by their massive parallelism, as described by Hager et al. [20], when the problem to be solved fits their features well.

NVIDIA GPU Architecture

From a hardware viewpoint, a GPU compatible with CUDA (Compute Unified Distributed Architecture) is made up of a scalable array of multithreaded Streaming Multiprocessors (SMs), each of which is able to execute several thread blocks at the same time. Each SM embeds eight scalar processing cores and is equipped with a number of fast 32-bit registers, a parallel data cache shared among all cores, a read-only constant cache and a read-only texture cache accessed via a texture unit that provides several different addressing/filtering modes. In addition, SMs can access local and global memory spaces which are (non-cached) read/write regions of device memory: these memories are characterized by latency times about two orders of magnitude larger than the registers and texture cache. Only threads belonging to the same thread block can share data in fast memory; different thread blocks may only share data allocated in slow memory. CUDA's scheduler allocates as many thread blocks at the same time as possible, compatibly with available resources, which allows a CUDA program to be run on any number of SMs. SMs can manage hundreds of threads running different code segments thanks to an architecture called SIMT (Single Instruction, Multiple Thread) which creates, manages, schedules, and executes groups (warps) of 32 parallel threads. Opposite to what happens in a SIMD (Single Instruction, Multiple Data) architecture, the whole execution and branching behavior of threads is specified. This way it is possible to manage parallel code for independent scalar threads as well as code for parallel data processing, which is executed by coordinated threads.

CUDA Programming Model

CUDA is a GPGPU environment, that includes a parallel computing architecture and programming model, developed by nVIDIA. This programming model requires the problem under consideration be partitioned into sub-problems, that are solved independently in parallel by blocks of threads. In turn, each sub-problem is also partitioned into finer pieces, that can be solved cooperatively in parallel by all threads within the same block. Blocks are organized into a one-dimensional, two-dimensional, or three-

dimensional grid of thread blocks, as illustrated in Figure 2.4.

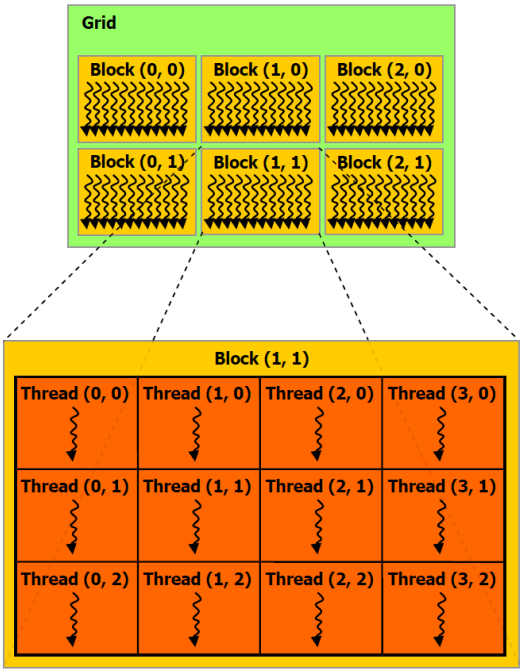


Figure 2.4: Grid of Thread Blocks [21]

Kernels

The programming language used within CUDA, CUDA-C, is an extension of the C programming language which allows one to implement GPU-based parallel functions, called kernels, which, when called, are executed N times in parallel by N different CUDA threads. Kernels are run on the device (GPU), while the rest of the code runs on the host (CPU), see Figure 2.5. It is also important to notice that, in CUDA, host and devices have separate memory spaces and, in order to execute a kernel, the programmer needs to explicitly allocate memory on the device and, if needed, transfer data from and back to the host. This is the main bottleneck which is encountered when optimizing code for speed. The programmer should reduce as much as possible the amount of these transfers.

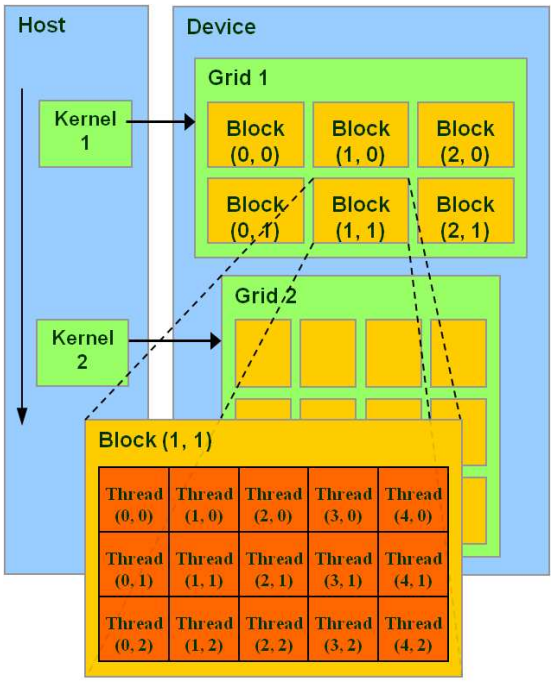


Figure 2.5: Kernel Execution [21]

Memory Hierarchy

The last thing to highlight is the memory hierarchy available to threads, and the performance associated with the read/write operations to/from each of the memory levels. Each thread has its own local *registers* and all threads belonging to the same thread-blocks can cooperate through *shared memory*. Registers and shared memory are physically embedded inside SMs and provide threads with the fastest possible memory access. Their lifetime is the same as the thread-block's. All the threads of a kernel can also access *global memory* whose content persists over all kernel launches [21], in addition to read-only *constant memory* and *texture memory*, which are located within the same memory space as the global memory; however, read and write operations to global memory are orders of magnitude slower than those to shared memory and registers, therefore access to global memory should be minimized within a kernel. The nVIDIA memory hierarchy is shown in Figure 2.6.

In order to obtain the best from this architecture, a number of specific program-

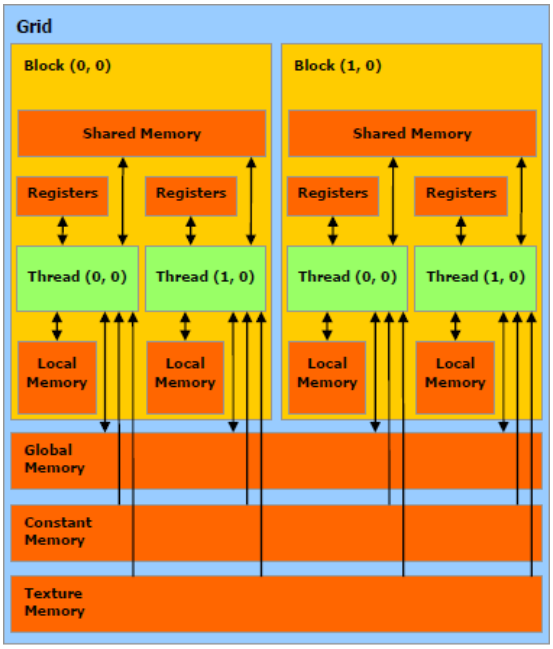


Figure 2.6: Memory Hierarchy and Access [21]

ming guidelines should be followed, the most important of which are: (a) minimize data transfers between the host and the graphics card; (b) minimize the use of global memory: shared memory should be preferred; (c) ensure global memory accesses are coalesced whenever possible; (d) avoid different execution paths within the same warp. Moreover, each kernel should reflect the following structure: (i) load data from global/texture memory; (ii) process data; and (iii) store results back to global memory.

An in-depth analysis of the architecture and more detailed programming tips can be found in [21, 22].

Chapter 3

Parallel Metaheuristics

Traditional optimization techniques, like the classical gradient search method, perform effectively when the problems under consideration satisfy tight constraints, such as being differentiable, convex and well-defined functions. However, when the search space is discontinuous, noisy, high-dimensional, and multimodal, then stochastic algorithms have been found to consistently outperform classical methods [23]. Among the stochastic approaches to continuous optimization, evolutionary and swarm intelligence algorithms, as well as other metaheuristics [24], offer a number of attractive features: no requirement for differentiable or continuous objective functions, robust and reliable performance, global search capability, virtually no need of specific information about the problem to solve, easy implementation, and implicit parallelism.

Despite several limitations which have been highlighted and the availability of other algorithms which perform better on global optimization benchmarks [25, 26], PSO and DE have recently become very popular. The main reason for their success is related to their associating good average performances with an easy implementation. However, the feature which is most relevant to our work and is shared with other evolutionary and swarm intelligence algorithms, is the fact that, being population-based and featuring limited dependency between each individual's operations, PSO and DE can be easily parallelized.

3.1 CUDA Particle Swarm Optimization

Parallel PSO seems to be the way to make practical use of this powerful search and optimization algorithm viable, in spite of its high computation cost. During the last decade, a considerable amount of literature about parallel PSO has been published. The first parallel PSO implementations relied on multiprocessor parallel machines or cluster computing systems [27, 28]. With the introduction of the GPUs, research shifted towards parallel PSO on the GPUs to alleviate multi-processor and cluster systems inefficiencies, such as network overhead, shared memory access, etc. Li et al. took advantage of GPU acceleration for developing parallel versions of PSO and GA through texture manipulation using shaders which are mainly used for graphics rendering purposes [29]. In 2009 de Veronese and Krohling developed the first implementation of PSO using nVIDIA CUDA [30].

Now that PSO can run efficiently on consumer-level graphics cards, researchers have experimented with new variants of the algorithm. Zhou and Tan extended the standard PSO to include the notion of 'unhealthiness' to describe swarms or sub-swarms stuck at local optima, then applying random mutations to the unhealthy particles' positions [31]. Also, Zhou and Curry created a hybrid between GPU PSO and pattern search to enhance the convergence of PSO [32].

Almost all recent GPU implementations assign one thread to each particle [30, 31, 33, 34] which, in turn, means that fitness evaluations have to be computed sequentially in a loop within each particle's thread. Since fitness calculation is often the most computation-intensive part of the algorithm, the execution time of such implementations is affected by the complexity of the fitness function and the dimensionality of the search domain. The speedup achieved by these implementations is evaluated with respect to their sequential counterparts executing on the CPU.

In addition, state of the art research in GPU-based parallelization of PSO focuses on the synchronous version of the algorithm, while it was shown, on distributed or cluster systems, that asynchronous versions can achieve faster execution time without sacrificing numerical accuracy [28, 35]. The asynchronous GPU PSO we present in the

following subsection overcomes the shortcomings of asynchronous PSO enforced by the master-slave approach used in distributed systems implementations, while gaining good speedup when compared to our synchronous GPU implementation [5] as well as, obviously, to the standard sequential PSO implementation.

CUDA Asynchronous PSO

To achieve both the fastest execution time and the best performance, we designed a parallel version of the algorithm, as fine-grained as possible, without introducing explicit synchronization mechanisms among the particles' evolution processes [6]. A main feature that affects the search performance of PSO is the strategy according to which the social attractor is updated. In 'synchronous' PSO, positions and velocities of all particles are updated one after another in turn during a 'generation'; this is actually a full algorithm iteration, which corresponds to one discrete time unit. Within the same generation, after velocity and position have been updated, each particle's fitness, corresponding to its new position, is evaluated. The value of the social attractor is only updated at the end of each generation, when the fitness values of all particles in the swarm are known. The 'asynchronous' version of PSO, instead, allows the social attractors to be updated immediately after evaluating each particle's fitness, which causes the swarm to move more promptly towards newly-found optima. In asynchronous PSO, the velocity and position update equations can be applied to any particle at any time, in no specific order. Regarding the effect of changing the update order or allowing some particles to be updated more often than others, Oltean and coworkers [36] have published results of an approach by which they evolved the structure of an asynchronous PSO algorithm, designing an update strategy for the particles of the whole swarm using a genetic algorithm (GA) and showing empirically that the GA-evolved PSO algorithm performs similarly, and sometimes even better, than standard approaches for several benchmark problems. Regarding the structure of the algorithm, they also indicate that several features, such as particle quality, update frequency, and swarm size, affect the overall performance of PSO [37].

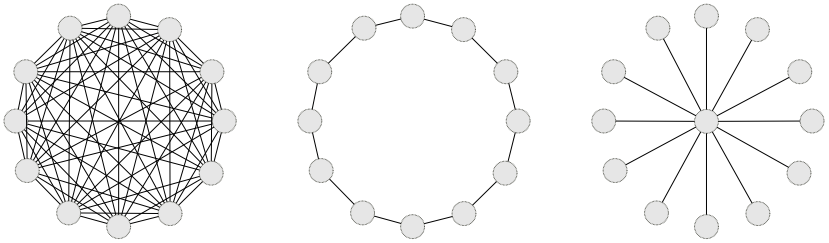


Figure 3.1: The three PSO topologies tested in this work. From left to right, global, ring and star

Implementation

As reported in the previous section, GPU implementations of PSO which assign one thread per particle, despite being the most natural way of parallelizing the algorithm, do not take full advantage of the GPU power in evaluating the fitness function in parallel. The parallelization only occurs on the number of particles of a swarm and ignores the dimensions of the function. In our parallel implementations we designed the thread parallelization to be as fine-grained as possible; in other words, all independent sequential parts of the code are allowed to run simultaneously in separate threads.

Swarm intelligence techniques are intrinsically parallel, because every swarm member has few dependencies on the others, so all operations aimed at adapting an individual's values, like position update or fitness evaluation, can be executed with few (or none at all) interactions with the other swarm members. From this point of view, in PSO the only data to be shared among particles is the global best position BGP visited so far by any member of the swarm, or the local best position BGP_i reached by the best fitness particle in the local neighborhood of particle i. Since the global best positions is the only information shared between particles, it has to be stored in global memory; the number of global memory reads within a block (representing a particle) differs depending on the PSO topology used. Figure 3.1 depicts the topologies implemented here: the global best topology, ring topology, and the star topology. In the first and third, only one global memory read per dimension/thread is necessary, while in the second, each particle compares its fitness to its two neighbors' (left and right), resulting in two global memory reads.

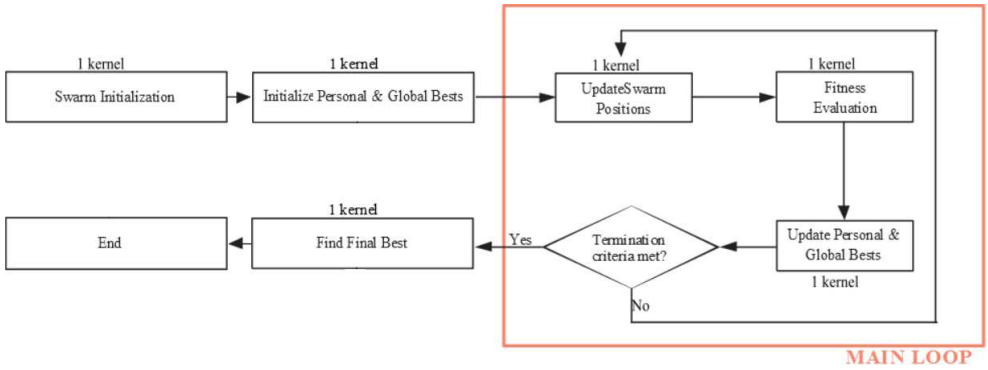


Figure 3.2: Block Diagram of the Synchronous CUDA PSO algorithm

CUDA kernels are executed sequentially, as shown in Figure 2.5, unless streaming is used. Consequently, the number of kernels used to implement a parallel algorithm greatly influences its performance. In Figure 3.2, we show the block diagram of the CUDA Synchronous PSO algorithm. First, all swarm particles are initialized to random positions within the search domain, using the nVIDIA CUDA Random Number Generation library (CuRAND) [38]. Then, a kernel evaluates the fitness of the random positions, and fills the best fitnesses and best positions global memory arrays. The main iteration loop of the algorithm consists of three kernels: particle positions update following the equations provided in Chapter 2, parallel fitness evaluation (some parts might be executed sequentially, depending on the nature of the fitness function), and the last kernel deals with deciding personal, global, or local best fitness values and positions, through a parallel reduction operation, which depends on the actual PSO topology employed. Finally, after a termination criteria has been met, often the generation/iteration maximum number, another kernel decides the final output of the optimization, through a parallel reduction to minimum/maximum operation.

To better understand the difference between synchronous and asynchronous PSO, the pseudo-code of the sequential versions of the algorithms are presented in Table 3.1. The synchronous 3-kernel implementation of CUDA-PSO, while allowing for virtually any swarm size, requires synchronization points where all the particles data have to be saved to global memory to be read by the next kernel. This frequent access to global memory limits the performance of synchronous CUDA-PSO and is the main

justification behind the asynchronous implementation.

Synchronous PSO	Asynchronous PSO

<Initialize positions/velocities of all particles>
<Set initial personal/global bests>
for(**int** i = 0; i < generationsNumber; i++)
{
for(**int** j = 0; j < particlesNumber; j++)
{
<Evaluate the fitness particle j>
}
<Update the position of all particles>
<Update all personal/global bests>
}
<Retrieve global best information to be returned as final result>
	<Initialize positions/velocities of all particles>
	<Set initial personal bests>
	for(**int** i = 0; i < generationsNumber; i++)
	{
	for(**int** j = 0; j < particlesNumber; j++)
	{
	<Evaluate the fitness of particle j>
	<Update the position of particle j>
	<Update personal bests of particle j>
	}
	}
	<Calculate global best information to be returned as final result>

Table 3.1: Pseudo-code for the sequential versions of PSO

The design of the parallelization process for the asynchronous version is the same as for the synchronous one, that is: we allocate a thread block per particle, each of which executes a thread per problem dimension. This way every particle evaluates its fitness function and updates position, velocity, and personal best for each dimension in parallel.

The main effect of the synchronization constraint removal is to let each particle evolve independently of the others, which allows it to keep all its data in fast-access local and shared memory, effectively removing the need to store and maintain the global best in global memory. In practice, every particle checks its neighbours' personal best fitnesses, then updates its own personal best in global memory only if it is better than the previously found personal best fitness. This can speed up execution time dramatically, particularly when the fitness function itself is highly parallelizable. This is a feature which often characterizes fitness functions which are commonly used in several applications, such as the squared sum of errors over a data set in classification tasks, or other fitness functions which can be expressed as a vector dot product or matrix multiplication.

In contrast to the synchronous version, all particle thread blocks must be executing simultaneously, i.e., no sequential scheduling of thread blocks to processing cores is employed, as there is no explicit point of synchronization of all particles. Two dia-

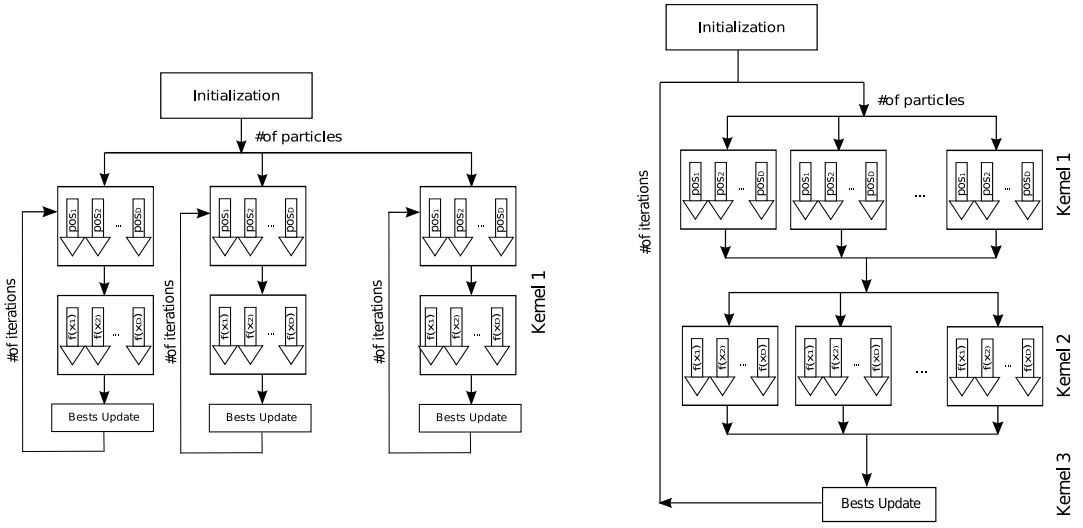


Figure 3.3: Asynchronous CUDA-PSO: particles run in parallel independently (left). Synchronous CUDA-PSO: particles evaluate fitness in parallel but have to wait the end of the generation before updating positions, velocities, and personal/global bests (right). Blocks represent particles and white arrows represent threads for each dimension of the search space.

grams representing the parallel execution for both versions are shown in Figure 3.3. Having the swarm particles evolve independently not only makes the algorithm more biologically plausible, as it better simulates a set of very loosely coordinated swarm agents, but it also does make the swarm more 'reactive' to newly discovered minima/maxima. The price to be paid is a limitation in the number of particles in a swarm which must match the maximum number of thread blocks that a certain GPU can maintain executing in parallel. This is not such a relevant shortcoming, as one of PSO's nicest features is its good search effectiveness; because of this, only a small number of particles (a few dozens) is usually enough for a swarm search to work, which compares very favorably to the number of individuals usually required by evolutionary algorithms to achieve good performance when high-dimensional problems are tackled. This consideration makes the availability of swarms of virtually unlimited size and the deriving potential in terms of search capabilities less appealing than it could seem at first sight, while increasing the relevance of the burden imposed, in terms of execution time, by the sequential execution of fitness evaluation. On the other hand, currently,

parallel system processing chips are scaling according to Moore's law, and GPUs are being equipped with more processing cores with the introduction of every new model.

3.2 CUDA Differential Evolution

The earliest CUDA implementation, up to our knowledge, of DE was presented in 2010 by [39]. After that, other implementations have been developed [32, 40], addressing problems with that first parallel version. In [39], the fitness evaluation, which is usually the most time consuming process, is performed in part sequentially, in the form of loops inside the device code (nested in case of mutation and crossover). Our fitness evaluation scales the number of working threads to the number of dimensions, calculating every dimension in parallel. We also use one block per solution/individual, eliminating the need for loops. Another problem with [39] is that they generate and store random numbers on the CPU for mutation, while we generate them on the fly on the GPU using the nVIDIA CuRAND library. In another DE implementation [41], four kernels are executed sequentially limiting the method's parallelization, while we implement one kernel for generating the trial vectors, and another for their fitness evaluation and migration. In addition, we offer three different mutations strategies and two kinds of crossovers, while early GPU-based DE considered only one mutation strategy (DE/rand/1) and one kind of crossover.

Implementation

PSO is divided into three kernels described in the previous section, while DE, as mentioned earlier, can be implemented as two kernels. Each thread of the first kernel performs the following instructions:

- generate two or three distinct random numbers on the GPU, according to the mutation strategy;

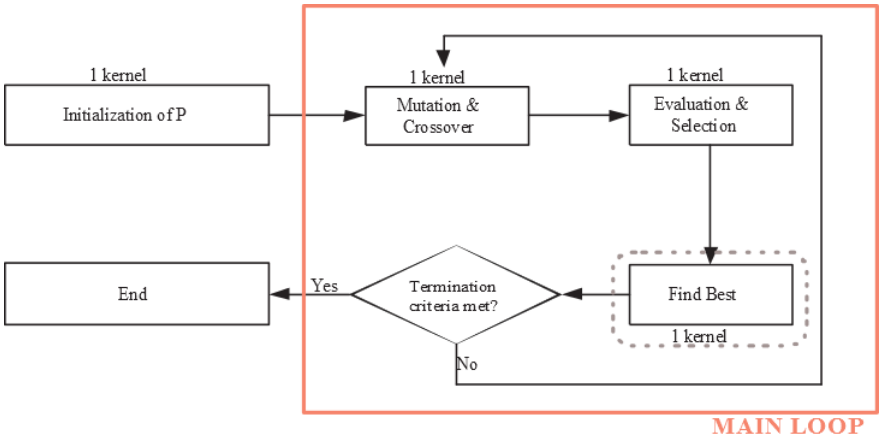


Figure 3.4: Block Diagram of the CUDA DE algorithm

- calculate an element of the donor vector from the population members randomly selected in the previous step;
- decide whether to include the donor or the parent element in the trial vector, based on the type of crossover and the crossover rate, Cr.

The second DE kernel evaluates all trial vectors simultaneously in shared memory and, if the fitness has improved, it replaces the parent with the offspring. In the cases of mutation "to-best" strategies, a third reduction kernel is needed to find the best individual, as highlighted in Figure 3.4.

3.3 CUDA Scatter Search

To the best of our knowledge, ours is the first parallel implementation of this meta-heuristic. Since Scatter Search is only a template for combining a global search method with an additional step of local search solution refinement [42], there are many variants of the algorithm, differing in the building blocks of this template. The most notable one is presented in [43], where the authors chose the Tabu Search local search method [44] for the refinement phase, mainly because of its adaptive memory capabilities, that are employed in order to remember previously visited/evaluated solutions in the local

neighborhood of a candidate solution to be refined.

Implementation

Clearly, SS is not as inherently parallel as the two other metaheuristics (see Figure 3.5). In SS a diverse population is first initialized and evaluated; diversity is simulated by generating uniform random values for each dimension over the whole search space. Then, to build the reference set R, a parallel sort operation is required to find R_1, followed by another kernel that calculates pairwise Euclidean distances between solutions in $P - R$ and R, sequentially adding the solutions that are farthest from the reference set for $|R_2|$ iterations. As for selection and crossover, a kernel selects all solution pairs in the reference set for mating, and combines them through the BLX-α crossover [45], generating two distinct solutions chosen with α set to $(0.5 + \lambda)$ and $(0.5 - \lambda)$, respectively. The combined solutions make up the *pool*, to which a parallel implementation of the Solis & Wets search method [13] is then applied as an improvement method. For the last step, we compared two methods for updating the reference set, one of which considers both quality and diversity as in [46], while the other updates the reference set with the best $|R|$ solutions in $(R \cup pool)$. The latter yielded better results in terms of both speed and accuracy.

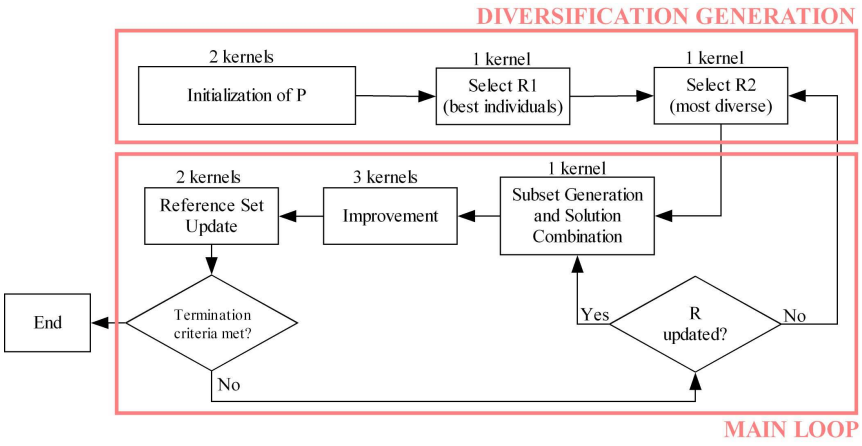


Figure 3.5: Block diagram of the Scatter Search Algorithm.

3.4 libCudaOptimize

libCudaOptimize [47] is an open source library which implements some metaheuristics for continuous optimization: presently Particle Swarm Optimization (PSO), Differential Evolution (DE), Scatter Search (SS), and Solis&Wets local search. This library allows users either to apply these metaheuristics directly to their own fitness function or to extend it by implementing their own parallel optimization techniques. The library is written in CUDA-C to make extensive use of parallelization, as allowed by Graphics Processing Units.

The main idea behind the library is to offer a user the chance to apply metaheuristics as simply and fast as possible to his own problem of interest, exploiting the parallelization opportunities offered by modern GPUs as much as possible. To the best of our knowledge, there are no software tools in which the entire optimization process, from exploration operators to function evaluation, is completely developed on the GPU, and allows one to develop both local and global optimization methods.

Implementation

libCudaOptimize is entirely written in C++ and CUDA-C and relies on two classes: `IOptimizer` and `SolutionSet` (see Figure 3.6). The former is an abstract class that includes all methods used for evolving a set of solutions (or population/swarm, where every particular solution is an individual/particle, depending on the used terminology), for setting evolution parameters and a reference to the set (it can evolve more than one set in parallel), represented by an instance of the class `SolutionSet`. Every different metaheuristic is implemented as a sub-class of `IOptimizer`. All these classes (see some examples at the bottom of Figure 3.6) have methods that allow a user to set the parameters of the metaheuristic. Moreover, most of the relevant parameters can be passed to the optimizer at the moment of its instantiation.

The class `SolutionSet` represents one or more sets of solutions and can be accessed in the user-defined fitness function, where it is used to access the elements

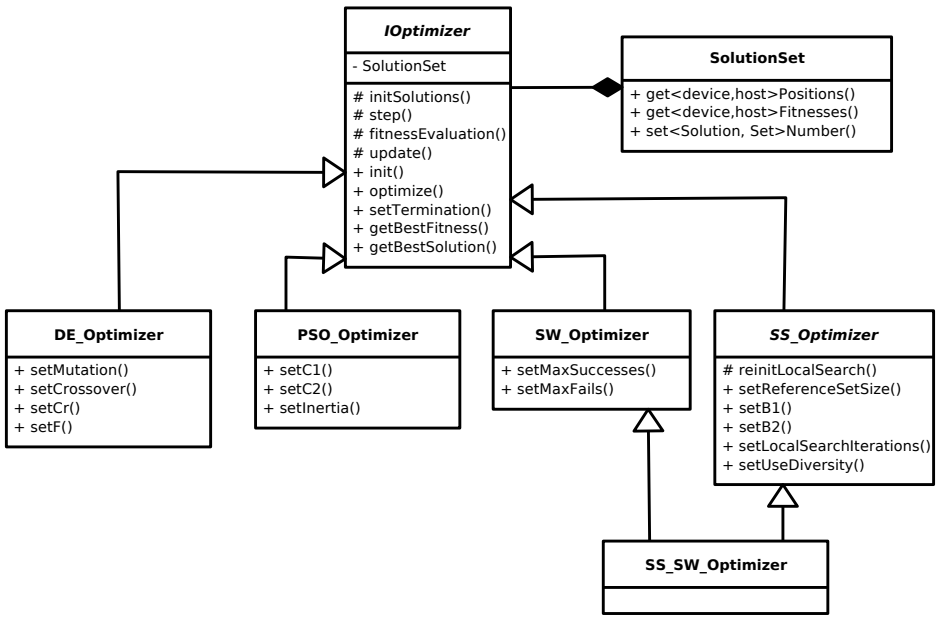


Figure 3.6: UML diagram. For every class, the most important methods are shown

of the population and to update their fitnesses after evaluation. There are methods that allow users to access the solutions, and their corresponding fitnesses, both on the device and the host. In this way, the user can employ these information both on C++ and CUDA-C function easily.

Usage

libCudaOptimize allows users to run their methods in parallel to optimize a fitness function, introduce a new optimization algorithm, or easily modify/extend existing ones. In the first case, the only thing one needs to do is to write the new fitness function in C++ or CUDA-C, while in the second and third cases, one can take advantage of the framework offered by the library to avoid the need to go deep into basic implementation issues, especially regarding parallel code.

libCudaOptimize is expected to be used by users who have, at least, a basic knowledge of C++. Although no explicit understanding of CUDA-C or even of metaheuristics is required it is very useful anyway; nonetheless, one can use this library just by

writing a C++ fitness function and launching one of the optimization techniques already implemented (to date PSO, DE, SS and Solis&Wets local search (SW)). This allows one to:

- implement commonly successful techniques with limited efforts;
- easily compare the results obtained by running different techniques on different functions;
- analyze the effects of changing values of the parameters which regulate the behavior of the optimization techniques on user-defined problems;
- run high-dimensional optimization experiments on consumer-level hardware, thanks to the efficient CUDA-C parallel implementation.

Basically, there are two ways to use this library. The first and most direct one is just to apply the included heuristics to optimize a user-defined fitness function. All one needs to do, in this simplest case, is to write a function in C++ or, to fully exploit the parallelization potentiality of the package, in CUDA-C. Then, one must select the heuristic, pass it the fitness function pointer, set its parameters, run it, and retrieve the solution(s) found.

The second purpose of the library is to allow the user to design and implement an optimization technique, taking advantage of the structure of the algorithms implemented in libCudaOptimize. Since several EAs share a similar structure, one can extend the superclass `IOptimizer` or one of its children in order to create a new optimizer. To do so, a mandatory step is to implement the four protected functions of `IOptimizer` shown in Figure 3.6:

- `initSolutions` randomly initializes the candidate solutions within the search space;
- `step` defines how the optimizer generates new potential solutions from the current population;

- `fitnessEvaluation` calls the user's fitness function;
- `update` is called after fitness evaluation and should update the population according to the results obtained: replace current individuals, update personal best locations, check constraints, ...

It is important to note that the user does not have to handle memory allocations and releases nor grid and kernels configuration, since these operations are taken care for by the library core.

3.5 Testing and Results

The parallel metaheuristics discussed in this chapter were tested against many theoretical and real-world applications. In [48, 49], both CUDA PSO and DE were successfully used to localize histological brain structures, in this case the hippocampus, and to estimate human body posture from multi-view video sequences, respectively. They were also the tool used in [50] for real-time traffic sign detection in sequences taken from a camera mounted on-board a car, and achieved good results in terms of quality and speed. This section will focus on the tests performed on benchmark functions, and consider the human body pose estimation problem as a case study on the usage of GPU metaheuristics in real-world applications.

Speedup Results

We compared the performance of the different versions of our parallel PSO implementation and of one sequential implementation based on the so-called Standard PSO (SPSO) [51] on a 'classical' benchmark which comprised a set of functions which are often used to evaluate stochastic optimization algorithms. Our goal was to compare different parallel PSO implementations with one another, and with a sequential implementation, in terms of speed. Since there is only a Standard version of PSO, in these tests we only focused on PSO. So we kept all algorithm parameters equal in

all tests, setting them to the 'standard' values suggested in [51]: $w = 0.729844$ and $C_1 = C_2 = 1.49618$. Also, for the comparison to be as fair as possible, we adapted the SPSO by substituting its original stochastic-star topology with the same ring topology adopted in the parallel GPU-based versions and we downgraded it to 'float' precision to match the GPU-based algorithms' precision.

The following implementations of PSO have been compared: (1) the sequential SPSO version modified to implement a two nearest-neighbors ring topology; (2) the synchronous three-kernel version of *CUDA-PSO*; (3) *CUDA-PSO* implemented asynchronously with only 1 kernel as in [6]. Values were averaged over the 98 best results out of 100 runs.

Figures 3.7 compares average execution times obtained for problem dimension D ranging from 2 to 128 in optimizing fitness functions from typical test-beds for function optimization. We tested our code on the following functions: (a) the simple Sphere function within the domain $[-100, 100]^D$, (b) Rastrigin function, on which PSO is known to perform well, within the domain $[-5.12, 5.12]^D$, (c) the Rosenbrock function, which is non-separable and thus hard to solve by PSO, within the domain $[-30, 30]^D$, and (d) the Griewank function within the domain $[-600, 600]^D$.

In general, the asynchronous version was much faster than the synchronous version, at the price of being able to run swarms of sizes up to 27 or 32 depending on the graphics card. It is also worth noticing that the execution time graphs are virtually identical for the functions taken into consideration, which shows that GPUs are extremely effective at computing arithmetic-intensive functions, mostly independently of the set of operators used, and that memory allocation issues are prevalent in determining performance. Taking speed-up values into consideration, one can also notice that the best performances were obtained on the Rastrigin and Griewank functions. This is probably due to the presence of complex math functions in their definition. In fact, GPUs have internal *fast math* functions which can provide good computation speed at the cost of slightly lower accuracy, which causes no problems in this case.

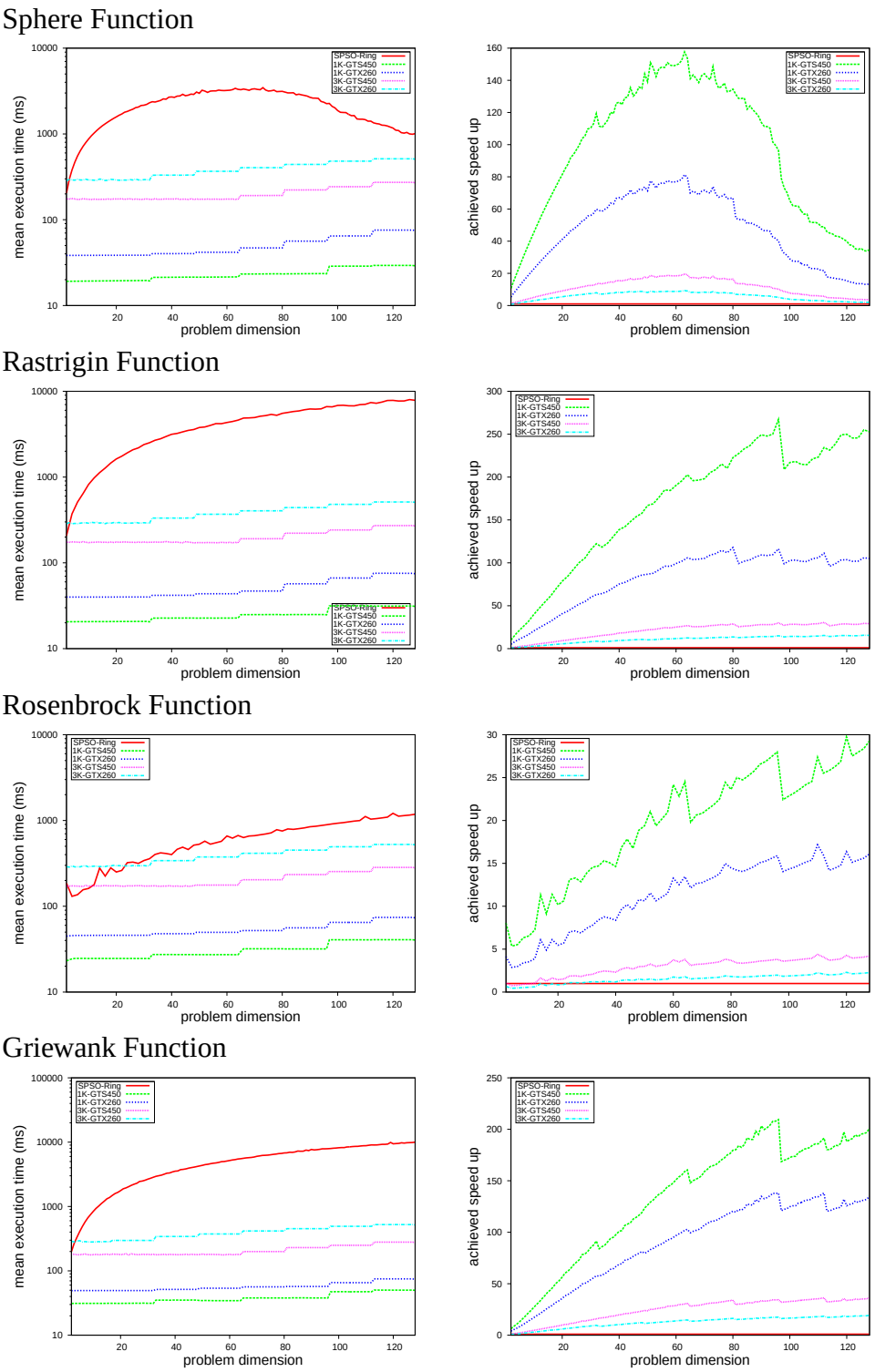


Figure 3.7: Average execution times (left column) and speed-ups (right column) vs. problem dimension for the Sphere, Rastrigin, Rosenbrock and Griewank functions (top to bottom). Experiments were performed running one swarm of 32 particles (GTS-450) or 27 (GTX-260) for 10000 generations. Plotted values were averaged over the best 98 results out of 100 runs.

Benchmark Functions

After establishing the superiority of parallel algorithms to their sequential counterparts, the subsequent experiments were run using methods implemented by means of libCudaOptimize, whiose results are reported in [52]. We evaluate both quality and speed of the parallel versions, analyzing their accuracy achieved in a limited amount of time, to assess the degree of parallelization that each of them allows to reach.

The algorithms we compared have a number of parameters that affect both accuracy and parallelism. "Manual" parameter tuning is time consuming and may introduce a bias in comparing an algorithm with a reference, due to better knowledge of the algorithm under consideration and to possible different time spent tuning each of them. Therefore, the automatic *tuning* of all three algorithms was performed using the `irace` software package [53], to find the configurations that yielded the best results in a given time: we set this time to one second, since it is generally short enough to avoid reaching full convergence with all three methods, allowing one to compare their short-term performances.

DE	PSO	SS
$Cr = 0.879$	$c_1 = 1.862$	$\|P\| = 140$
$F = 0.520$	$c_2 = 1.881$	$\|R_1\| = 9, \|R_2\| = 1$
Exponential Crossover	$w = 0.494$	$\lambda = 0.220$
Random Mutation Size = 48	Population Size = 125	Solis & Wets iterations = 85

Table 3.2: Automatically-tuned parameter values used to test different optimization techniques.

The tuner was run on all 20 functions with a budget of 30000 experiments, each being one run of one configuration on one function with a termination criterion of one second. Since the functions have different fitness ranges, a rank-based test is preferable to a test based on the solutions' mean values. Accordingly, the Friedman test was used to discard significantly worse configurations. We tuned the parameters for 30-dimensional problems, and assumed that such configurations are good also for lower-sized ones. Table 3.2 displays the parameters that have been tuned for each

algorithm, and the best corresponding values.

We compared our results to the values that are most commonly used in literature. For instance, the authors in [9] suggest $F \in (0.4, 0.95)$ and $Cr \in (0.9, 1)$ for multimodal separable functions (the most common ones in our benchmark); we obtained similar results. Regarding PSO, in most papers, $c_1 = c_2 = 2.0$ [54], while our automatic tuning set them to slightly smaller values.

	Name	Range	Formula		
f_0	Sphere	$[-100, 100]^n$	$\sum_{i=0}^{n-1} x_i^2$	U	S
f_1	Elliptic	$[-100, 100]^n$	$\sum_{i=0}^{n-1} (10^6)^{\frac{i-1}{D-1}} x_i^2$	U	S
f_2	Sum of Squares	$[-1, 1]^n$	$\sum_{i=0}^{n-1} i x_i^2$	U	S
f_3	HyperEllipsoid	$[-1, 1]^n$	$\sum_{i=0}^{n-1} i^2 \cdot x_i^2$	U	S
f_4	Schwefel 2.22	$[-10, 10]^n$	$\sum_{i=0}^{n-1} \lvert x_i \rvert + \prod_{i=0}^{n-1} \lvert x_i \rvert$	U	S
f_5	Zakharov	$[-10, 10]^n$	$\left(\sum_{i=0}^{n-1} {x_i}^2\right) + \left(\sum_{i=0}^{n-1} 0.5 \cdot i \cdot {x_i}^2\right)^2 + + \left(\sum_{i=0}^{n-1} 0.5 \cdot i \cdot {x_i}^2\right)^4$	U	S
f_6	Schwefel 1.2	$[-100, 100]^n$	$\sum_{i=0}^{n-1} \left(\sum_{j=0}^{i} x_j\right)^2$	U	NS
f_7	Schwefel 2.6	$[-100, 100]^n$	$max\{\mathbf{A}_i\mathbf{x} - \mathbf{B}\}$, $i = 0, \ldots, n-1, \mathbf{x} = [x_0, \ldots, x_{n-1}]$, $\mathbf{A}_i, \mathbf{B}$ defined in [55].	U	NS
f_8	Dixon-Price	$[-10, 10]^n$	$(x_0 - 1)^2 + \sum_{i=1}^{n-1} \left(i \cdot \left(2{x_i}^2 - x_{i-1}\right)^2\right)$	U	NS
f_9	Rastrigin	$[-5.12, 5.12]^n$	$\sum_{i=0}^{n-1} \left\{x_i^2 - 10 \cdot \cos(2\pi x_i) + 10\right\}$	M	S
f_{10}	Schwefel 2.26	$[-500, 500]^n$	$418.9829 \cdot n + \sum_{i=0}^{n-1} \left(x_i \cdot \sin\sqrt{\lvert x_i \rvert}\right)$	M	S
f_{11}	Katsuura	$[-1000, 1000]^n$	$\prod_{i=0}^{n-1} \left(1 + (i+1) \sum_{k=1}^{d} round(2^k x_i) 2^{-k}\right) - 1$	M	S
f_{12}	Griewank	$[-600, 600]^n$	$\sum_{i=0}^{n-1} \frac{x_i^2}{4000} - \prod_{i=0}^{n-1} \cos(\frac{x_i}{\sqrt{i}}) + 1$	M	NS
f_{13}	Rosenbrock	$[-100, 100]^n$	$\sum_{i=0}^{n-1} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$	M	NS
f_{14}	Ackley	$[-32, 32]^n$	$-20e^{-0.2\sqrt{\frac{1}{n}\sum_{i=0}^{n-1} x_i^2}} - e^{\frac{1}{n}\sum_{i=0}^{n-1}\cos(2\pi x_i)} + 20 + e$	M	NS
f_{15}	Griewank + Rosenbrock	$[-5.12, 5.12]^n$	$f_{griewank}(f_{rosenbrock})$	M	NS
f_{16}	Scaffer	$[-100, 100]^n$	$\sum_{i=0}^{n-1} F(x_i, x_{i+1}), x_n = x_0$ where $F(x, y) = 0.5 + \frac{\sin^2\left(\sqrt{x^2+y^2}\right) - 0.5}{1 + 0.0001(x^2+y^2)}$	M	NS
f_{17}	Schwefel 2.13	$[-\pi, \pi]^n$	$\sum_{i=0}^{n-1} \left(\mathbf{A}_i - \mathbf{B}_i(\mathbf{x})\right)^2, \mathbf{x} = [x_0, \ldots, x_{n-1}]$ $\mathbf{A}_i, \mathbf{B}_i(x)$ defined as in [55].	M	NS
f_{18}	Salomon	$[-10, 10]^n$	$-\cos\left(2\pi\sqrt{\sum_{i=0}^{n-1} {x_i}^2}\right) + 0.1\sqrt{\sum_{i=0}^{n-1} {x_i}^2} + 1$	M	NS
f_{19}	Levy	$[-10, 10]^n$	$\sin^2(\pi y_0) + \sum_{i=0}^{n-2} \left[(y_i - 1)^2 \left(10\sin^2(\pi y_i + 1)\right)\right] + (y_{n-1} - 1)^2 \left(1 + 10\sin^2(2\pi y_{n-1})\right)$ where $y_i = 1 + \frac{x_i - 1}{4}, i = 0, \ldots, n-1$	M	NS

Table 3.3: Benchmark functions. For every function, the table shows the name, the range of the search space, the formula, the multimodality (multimodal, unimodal) and the separability (separable, non separable). All minima are in $\{0\}^n$.

To evaluate both the effectiveness and the efficiency of the three parallel implementations, tests on 20 numerical benchmark functions (see Table 3.3) were run on a 64-bit Intel(R) Core i7 CPU running at 2.67GHz using CUDA v. 4.1 on a nVIDIA GeForce GTS450 graphics card with 1GB of DDR memory and compute capability 2.1 [21]. Table 3.4 reports the results obtained executing 100 runs per function (6000 independent runs) and setting 1 second as the only termination criterion. The first column is the function under consideration. The following ones are divided into two blocks according to the number of dimensions (10 and 30). Within each block, the mean best fitness and the standard deviation over all runs are reported for each method. Results reported on a grey background highlight those cases in which the median over 100 runs obtained by the method is significantly better than the other methods, according to the Kruskal-Wallis test, with a confidence level of 0.01.

The results reported in Table 3.4 and Figure 3.8 allow one to draw some conclusions about the behaviour of the three parallel metaheuristics. Conforming with previous results obtained by sequential implementations, DE obtained the best results, sometimes tied with some other method, in 35 out of the 40 experiments performed, while PSO was the best method, sometimes tied with some other method, in 20 out of 40 functions, its main drawback being its tendency to stagnate and find sub-optimal solutions more often than DE, even if a higher number of function evaluations is run. Regarding SS, whose first parallel implementation is presented here, it obtained the best result in 11 out of 40 problems; however, this metaheuristic, which is not as parallelizable as the other methods, as reflected by the number of kernels, has achieved better performance over multimodal non-separable problems and time-consuming fitness functions, like Katsuura.

All tests were run with a temporal limit of one second, a short time in which all three methods can generally obtain results close to the optima without reaching full convergence. Figure 3.8 shows that PSO requirqes almost three times as many fitness function evaluations as DE to converge on 30-dimensional problems. It is important to notice that the population size in PSO is also almost three times as large as in DE, which justifies the larger number of fitness evaluations.

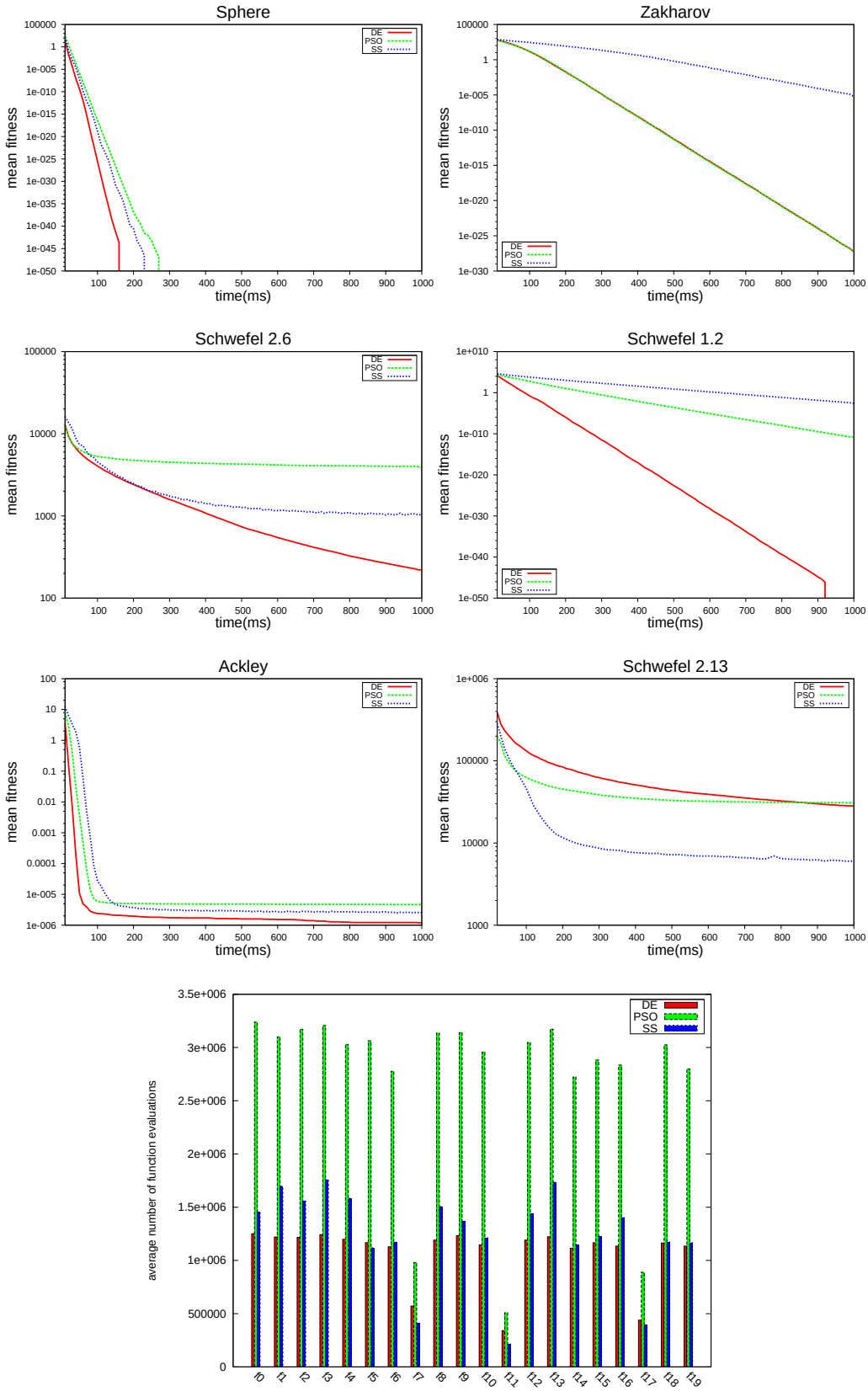


Figure 3.8: Mean fitness vs time (up to 1 second) for six representative functions (unimodal separable, unimodal non-separable and multimodal non-separable), and number of function evaluations performed in 1 second by every method for each function on 30-dimensional problems.

	10 dimensions						30 dimensions					
	DE		PSO		SS		DE		PSO		SS	
	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std	Avg	Std
f_0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
f_1	0.0	0.0	0.0	0.0	2.5e-03	9.2e-03	0.0	0.0	0.0	0.0	2.2e-06	6.3e-06
f_2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.1e-44	8.6e-44
f_3	0.0	0.0	0.0	0.0	7.0e-45	2.9e-44	0.0	0.0	0.0	0.0	6.7e-05	3.0e-04
f_4	0.0	0.0	0.0	0.0	1.1e-25	1.7e-25	0.0	0.0	0.0	0.0	5.1e-24	2.7e-24
f_5	0.0	0.0	0.0	0.0	0.0	0.0	2.5e-28	3.4e-28	1.8e-28	2.2e-28	1.2e-05	1.7e-05
f_6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	9.7e-012	9.9e-012	2.5e-03	3.3e-03
f_7	9.8e-06	6.9e-05	1.0e-03	4.0e-04	2.7e-04	3.5e-04	2.1e+02	3.3e+02	3.8e+03	1.1e+03	9.9e+02	3.4e+02
f_8	5.0e-01	5.0e-08	3.5e-02	1.3e-01	2.9e-01	2.5e-01	5.0e-01	0.0	4.2e-01	1.9e-01	5.0e-01	2.1e-05
f_9	0.0	0.0	5.2e-01	7.8e-01	6.9e-01	9.1e-01	0.0	0.0	7.2e+01	1.6e+01	3.5e+01	9.7e+00
f_{10}	5.9e+00	3.1e+01	1.2e+02	1.2e+02	8.1e+01	1.1e+02	1.7e+01	4.8e+01	2.9e+03	4.1e+02	2.4e+03	9.0e+02
f_{11}	1.2e-03	4.9e-06	1.2e-03	2.9e-05	1.2e-03	2.4e-18	1.2e-02	5.1e-05	1.2e-02	1.8e-04	1.2e-02	8.7e-18
f_{12}	0.0	0.0	1.1e-02	1.0e-02	1.1e-03	2.9e-03	7.4e-05	7.4e-04	6.3e-10	6.0e-09	1.5e-03	4.3e-03
f_{13}	0.0	0.0	3.9e-07	4.8e-07	5.9e-01	3.3e+00	0.0	0.0	2.1e-01	7.2e-01	2.2e+01	2.6e+01
f_{14}	0.0	0.0	6.7e-07	1.1e-06	0.0	0.0	1.1e-06	1.2e-06	4.5e-06	9.4e-07	9.3e-03	9.3e-02
f_{15}	0.0	0.0	1.2e-03	6.5e-03	6.3e-31	4.4e-30	0.0	0.0	1.5e-28	1.2e-27	1.1e-27	2.7e-27
f_{16}	3.3e-02	2.9e-02	1.0e-01	2.4e-02	7.3e-01	5.4e-01	3.2e-01	3.0e-02	8.5e+00	8.6e-01	8.7e+00	1.2e+00
f_{17}	4.5e+01	2.2e+02	1.3e+00	5.1e+00	4.7e+00	8.5e+00	2.8e+04	6.1e+03	3.1e+04	1.8e+04	5.2e+03	5.6e+03
f_{18}	1.0e-01	2.8e-17	1.0e-01	2.8e-17	9.8e-02	1.4e-02	1.9e-01	3.1e-02	2.0e-01	1.7e-02	2.4e-01	5.1e-02
f_{19}	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.3e-02	1.0e-01

Table 3.4: Results on the 20 benchmark functions.

Real-World Application

Moving more towards the application side, this section describes a real-world problem to which metaheuristics were applied, to both solve the problem and compare the effectiveness of DE and PSO as global optimization methods within this framework. The problem addressed here is that of the three-dimensional human body pose estimation. 3D human body pose estimation from video is the problem of extracting an accurate estimate of the posture of a human body, along with its location in space, from an image or a frame within a video stream. This is a complex problem that has been invariably formulated as a high-dimensional space search problem, due to the complexity of the human body pose parameterization. The problem has been tackled by trying to reduce the complexity of the search while also relying on effective search schemes.

The search complexity can be reduced based on local predictions, e.g., using particle filters [56, 57], or by partitioning the search space into smaller, more manageable subspaces [56, 58]. The use of machine learning techniques to define specific motion models for particular actions from training data collected in advance has also been considered [59, 60]. These approaches suffer from various setbacks. The particle-filtering solutions critically rely on a high number of particles to adequately represent the posterior distribution, which increases their computational complexity beyond practical use when considering a wide variety of motion. As well, relying on pre-trained motion models causes the human body tracking approaches to lose their generalization abilities, which points to methods that can reliably provide motion estimates without depending on much prior knowledge [61].

In [62, 63], an effective search algorithm was proposed, which is capable of recovering the pose without any prior knowledge of the nature of motion. The main drawback of the method is its huge computational complexity, which makes the time required for execution of a standard sequential implementation hardly acceptable. However, relying on the parallel nature of both the search algorithm and the multi-view pose estimation problem by implementing the approach on a graphical processing unit (GPU), the authors showed that they could reach execution times acceptable for practical purposes [49].

Model Formulation

The input consists of N views of the body, taken from different angles. From each image, we extract the silhouette of the body, i.e., a binary image in which all pixels belonging to the body are set to 1. The set of silhouettes represents the target to be matched to the silhouettes generated by a transformation of the model, according to the following steps:

- a pose estimation is generated by the search algorithm;
- a 3D rendering of the body, in such a pose, is made;
- a set of N images, corresponding to the projections of the rendered body (silhouettes) on the image planes of the input cameras, is computed.

The body model consists of two layers, the skeleton and the skin. The skeleton layer is defined as a set of homogeneous 4×4 transformation matrices which encode the information about the position and orientation of every joint with respect to its parent joint in the kinematic tree hierarchy. The skin layer, which represents the second layer in the model, is connected to the skeleton through the joints' local coordinate systems. Each joint controls a certain area of the skin. Whenever a joint or limb moves, the corresponding part of the skin moves and deforms with it. As the skin is a subdivision surface, only the base mesh has to be specified in the corresponding joint coordinate system. After the joint configuration has been specified, the base mesh is subdivided by repeatedly applying the Catmull-Clark subdivision operator [64] until the desired smooth shape of the body is obtained.

Considering the body composed of head, torso, and a three-joint kinematic chain for each limb, the model has 32 degrees of freedom, represented by real-valued parameters: three of them represent the global body position in space, while the other 29 represent relative angles, in space, between consecutive segments of the kinematic chains, i.e., joint orientations. These are subject to anatomical constraints which limit both their number and possible value range. A more detailed description of the model can be found in [49].

Fitness Function

The fitness function compares the silhouettes extracted from the original images to the silhouettes generated by the model in its candidate pose. Let the images containing the *original* silhouettes be denoted as I_i^o, $i = 1...N$. Similarly, let I_i^m, $i = 1...N$ denote the images of the *model* silhouettes. The cost function can then be written as follows:

$$E = \sum_{i=1}^{N} \frac{1}{Z_i} \sum_{1}^{row} \sum_{1}^{col} (I_i^o \ \& \ I_i^m), \tag{3.1}$$

where row and col denote the number of image rows and columns, respectively, and $\&$ denotes the bitwise *AND* operation. Coefficients Z_i are the normalization constants obtained by counting the number of silhouette pixels in every original image. Therefore, the fitness value that can be obtained for each view ranges from 0 to 1, with 0 corresponding to the absence of overlapping between the two silhouettes, and 1 to a perfect overlap. Thus, the overall fitness value (E) ranges from 0 to N.

Experiments and Results

Tests were run on a computer equipped with a 64-bit Intel® Core i7 CPU running at 2.80 GHz with 6 Gb of RAM using CUDA v. 4.1 on a nVIDIA GeForce GTS450 graphics card with 1GB of DDR memory and compute capability 2.1.

DE	PSO
$Cr = 0.9$	$c_1 = 2.0$
$F = 0.5$	$c_2 = 2.0$
Uniform Crossover	$w = 2.0/e^x$
Mutation: DE/rand/1	
Population Size = 10	Population Size = 10

Table 3.5: Parameters used for human body pose estimation. Regarding the inertia factor w, we set $x_{initial} = 2.0$; $x = x + 0.05$ if, at the end of a generation, the global best has not improved. For the very first frame $x_{initial}$ was set to 1.0 to increase the algorithm's exploration ability, when it is required to recover the initial pose from scratch.

We compared the results obtained by our CUDA implementations of DE and PSO. The parameter values used in the tests were set starting from the most commonly used values reported in literature, and refined during the development of the system. The values we set for the most relevant parameters are shown in Table 3.5.

Our algorithms were tested on a set of 4 test sequences, kindly made available by the CVSSP, University of Surrey. They were acquired in a dedicated multi-camera acquisition studio and consist of 10 synchronized videoclips with resolution 720×576, and a frame rate of 25 fps.

Since these sequences come with no ground truth, we decided to create a "synthetic" sequence to statistically estimate the error made by our system in recovering the pose of the body. To do so, we took the sequence containing the most complex (and fastest) movements, which represents a man performing a karate kick, and let our system optimize it multiple times for a very high number of generations. After collecting the best results (highest fitness values) for each frame, we rendered the silhouette images of our model in those very same positions. This way we obtained an artificially created sequence of which the articulated model we employ exactly matches all the silhouettes available and for which we know, frame by frame, the actual pose of each joint of the model. In other words, we created a synthetic sequence which comes with "ground truth" values for all the parameters we need to optimize. After this, we compared the three-dimensional position of every joint of the model in the reference sequence and the values obtained as output by the test runs of our method.

It is important to remark that, in the final tests, instead of setting a fixed number of iterations/generations as in most iterative algorithms, we used the value of the decreasing inertia parameter defined in Table 3.5 as a stopping criterion for both DE and PSO, ending our process when w fell below 0.1.

	Average	StdDev	Worst	Best	Median	Wilcoxon
DE	6.41	6.60	41.42	0.36	3.76	$< 1.0E-10$
PSO	4.32	4.47	32.71	0.22	2.40	-

Table 3.6: Results of human body pose estimation: average distance values (in cm) to the joints obtained processing the reference sequence in ten independent runs.

	Average	StdDev	Worst	Best	Median	Wilcoxon
DE	8.21	0.11	8.06	8.42	8.91	$< 1.0E-10$
PSO	8.24	0.13	8.05	8.45	8.98	-

Table 3.7: Results of human body pose estimation: average fitness values obtained processing the reference sequence in ten independent runs.

	Average	StdDev	Worst	Best	Median	Wilcoxon
DE	8.30	0.11	8.13	8.49	8.89	$< 1.0E-10$
PSO	8.34	0.12	8.15	8.52	8.94	-

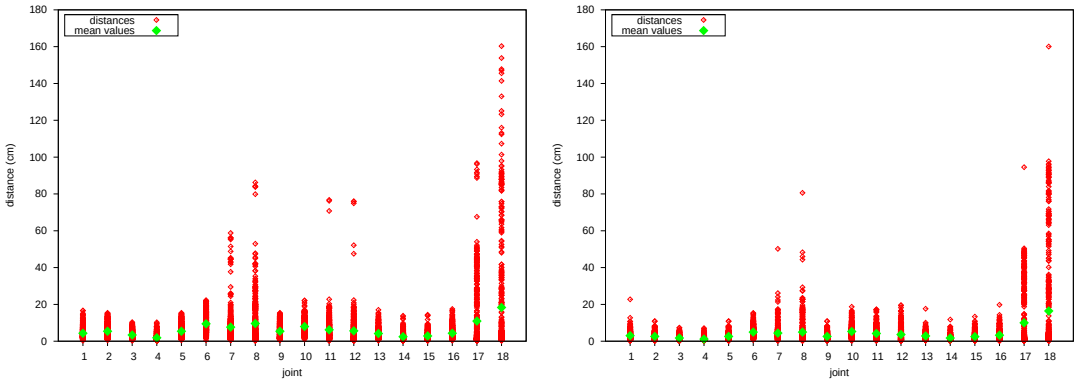
Table 3.8: Results of human body pose estimation: average fitness values obtained processing all the "real" video sequences in ten independent runs.

The first two tables refer to the results obtained processing the reference sequence. In particular, in Table 3.6 we show the results obtained by PSO and DE, expressed as distances, and in Table 3.7 as fitness values (higher fitness values are associated with better solutions). In Table 3.8 we show the global results as fitness values computed on the other four sequences.

The first column in all tables is the mean value of the measure under consideration over all runs and frames. For example, in Table 3.7, the value in the first row and first column is the fitness obtained by DE averaged over the 500 executions of the algorithm (50 frames and 10 runs). The second column reports the mean of the average standard deviations obtained for every joint in the model. The third and fourth columns report the mean of the worst and best values, respectively, averaged over all frames in each run. The fifth column is the mean of the median values for each run. Finally, the last column in all tables reports the p-value obtained with the Wilcoxon Signed-Rank test [65] with a significance level of 0.001.

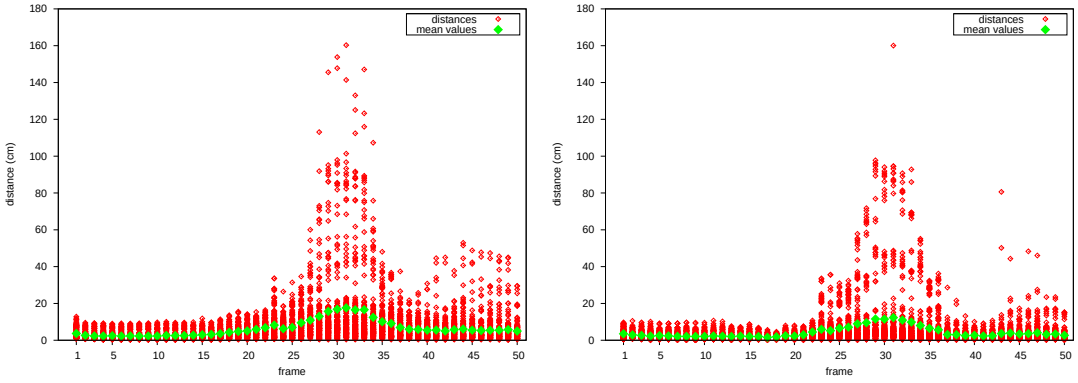
The null hypothesis used in Table 3.6 was that the median of distances obtained by PSO is greater or equal than the median of the distances obtained by DE. In Tables 3.7 and 3.8 the p-value refers to the following null hypothesis: the median fitness obtained by PSO is less or equal than the median fitness obtained by DE.

In Figures 3.9 and 3.10, all the results obtained, per joint and per frame respectively, are plotted.



(a) DE (b) PSO

Figure 3.9: Body pose estimation: per-joint performance on the reference video sequence. Scatter plot of the distances (in cm) of each joint from the ground truth estimated over all frames over 10 runs. Means are represented by lighter bullets.



(a) DE (b) PSO

Figure 3.10: Body pose estimation: per-frame performance on the reference video sequence. Scatter plot of the distances from the ground truth of all joint estimates over 10 runs for each of the 50 frames. Means are represented by lighter bullets.

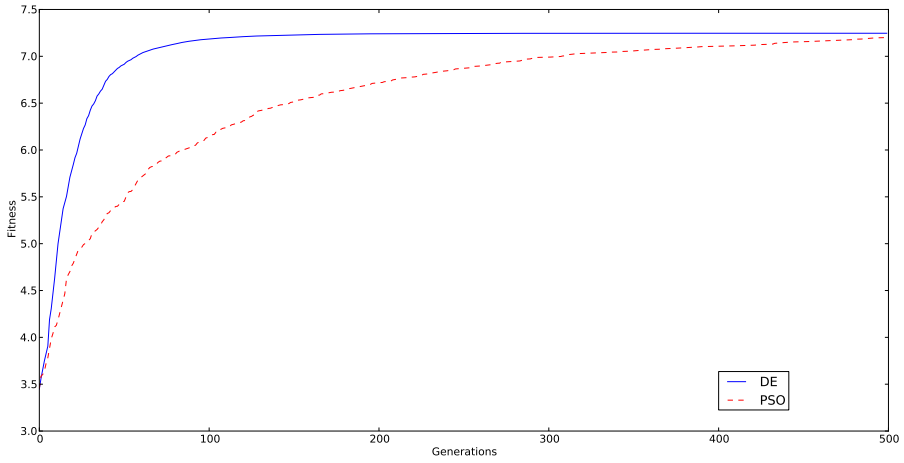


Figure 3.11: Average fitness values vs number of generations for PSO and DE in the body pose estimation problem for the first frame of the 5 sequences. Full optimization of the whole body.

In the first frame, the initialization of the swarm is completely random, while in the subsequent ones the swarm is initialized in a vicinity of the best pose found in the previous frame, thereby implementing some sort of tracking. As the pose changes only slightly between two consecutive frames, performing the optimization over the whole search space is both unnecessary and time consuming. However, to investigate the general localization ability of DE and PSO, the optimization was allowed to run for more iterations, 500 in this case, and the hierarchical optimization steps, described in [49], were also removed to increase the complexity of the problem. Results are reported in Figure 3.11, showing how fitness values improve during the optimization process. As explained before, only the results obtained processing the first frame are actually representative of the global search ability of the method used.

As the tables show, the results obtained by PSO in this problem using the hierarchical optimization are better than the ones obtained with DE, in terms of fitnesses and distances. However, if we increase the complexity of the problem by removing the hierarchical strategy and the time constraints, optimizing all the parameters at the same time, DE obtains better results than PSO (see Figure 3.11). This behavior can be

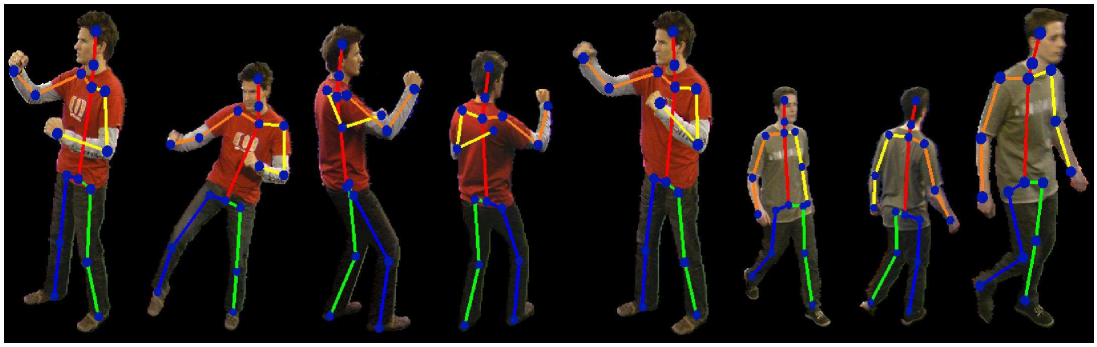


Figure 3.12: Example pose results shown as skeletons overlaid on the corresponding input image. The examples shown are taken from different sequences (JonWalk, Tony Kick, Tony Punch and Tony Stance) and different camera views (10 views were used for each sequence), hence the difference in person size as well as orientation.

explained, in first place, taking into account that DE is one of the best-performing methods for large-scale continuous optimization problems [9]; therefore, the more complex the environment the better the expected performance with respect to other methods. A second explanation can be the evolutionary nature of DE; since the scaled differences of randomly selected and distinct population members are combined to create new solutions, the weighted combination of good partial solutions can produce very good global results.

It is also important to notice that the best results are obtained using the hierarchical approach, in which, whenever a good position for one part of the body is found, all other joints are constrained to this newly found best position (i.e. they cannot explore other orientations and positions that are inconsistent with it). With respect to execution time, the hierarchical version of the human body pose estimation using DE takes on average 4677.60 ms per frame, while the corresponding version based on PSO takes 4810.33 ms. Figure 3.12 shows examples of estimated poses for different camera views and different sequences.

3.6 Final Remarks

In this chapter the rationale behind parallel metaheuristics was given, along with the details of our GPU implementations of some popular optimization techniques on the nVIDIA CUDA platform. The gains of parallel implementations of metaheuristics are manifold: exploiting the inherently parallel nature of those algorithms, which, in turn, allows the use of such powerful global optimizers on consumer level hardware in a fraction of the computing power and time required to run their CPU/sequential counterparts. We mainly implemented three methods: PSO, DE, and, SS, with two distinct variants of PSO: synchronous, and asynchronous PSO. The GPU-based asynchronous version of the PSO algorithm was able to significantly reduce execution time with respect to the synchronous one, imposing limitations on the number of particles which seemed not to affect performances significantly, at least on the benchmark we used for tests. Depending on the degree of parallelization allowed by the fitness functions we considered, the asynchronous version of CUDA-PSO could reach speed-ups of up to about 300 (in the tests with the highest-dimensional Rastrigin functions) with respect to the sequential implementation, and often of more than one order of magnitude with respect to the corresponding GPU-based 3-kernel synchronous version, sometimes showing a limited, possibly only apparent, decrease of search performances.

The development of libCudaOptimize, in addition to the automatic tuning abilities of `irace`, allowed us to make a fair comparison of our implemented GPU-based methods, chiefly due to the common code base that was abstracted by the library. The results reported in the tests comparing the performance of the implemented parallel metaheuristics, although conforming with the No Free Lunch theorem [1], did in fact prove the superiority of one of the methods, DE, in attaining the best overall results over the set of benchmark functions used in the experiments. However, the main purpose of the experiments was not to prove the superiority of one algorithm over the other in terms of general applicability, but rather in terms of how well the parallel version performs, under the given time constraint. Ultimately, the foreseen goal of libCudaOptimize is to expose the power of GPU metaheuristics to researchers and users from different fields of science.

Finally, we described a parallel approach to articulated human body pose estimation from multi-view video sequences, based on the CUDA architecture. The results show that the execution time can be cut down noticeably by formulating the algorithm on the GPU, without sacrificing the pose estimation accuracy, thereby exploiting the vast computational resources available on an ordinary desktop PC. The current implementation still combines the computational power of the CPU and GPU, for example, for the purpose of camera projection, which induces a computational overhead when passing data between the two processing units. Additional speedup is therefore possible by deploying the complete algorithm on GPU in order to avoid the communication bottleneck. This would also allow us to increase the size of the swarm, which is likely to lead to better performance. A further improvement is anticipated from exploiting the parallelism in the kinematic structure of the human body. Both improvements have been left as future work.

Chapter 4

Hierarchical Quilted Self Organizing Maps

The HQSOM was introduced in [66] mainly to model the vision part of the neocortex, and achieve biological similarity to a considerable extent. It builds upon previous computational models such as the Neocognitron [67], Neural Abstraction Pyramid[68], and VisNet [69]. However, it improves upon those models and HTM in several aspects. Firstly, it uses the same simple algorithm for both spatial and temporal clustering, the Self Organizing Map (SOM), and the Recurrent Self Organizing Map (RSOM), respectively. Furthermore, unlike the HTM, it employs online learning through the training and testing phases, thus adapting to new inputs and increasing generalization. Finally and more importantly, it uses temporal associations to form invariant representations of causes and patterns in spatio-temporal sequences, while models like the Neocognitron are more suited to recognition tasks from a single image, ignoring the time aspect that humans and animals rely on to evolve their sophisticated vision systems.

Much like the HTM, HQSOM tries to exploit the spatial correlations between pixels of an input image to form transformation invariant representations. Figure 4.1 shows a HQSOM configuration with a two dimensional grid as input, as is the case when working with pixels of an image or a video frame. As described earlier, each layer is composed of a number of units or nodes running exactly the same procedure;

in the case of HQSOM this unit is denoted as the *SOM-RSOM pair*. The input feature vector can be divided into overlapping or non-overlapping receptive fields, and fed to a single SOM-RSOM pair. This subdivision of the input space ensures that every pair only responds to features in its corresponding field, thus emulating the shift invariance property of biological vision.

Our HQSOM implementation follows the description above to an extent. However, as it stands, this system can not cope with real world applications. In fact, in the original paper, HQSOM was only tested with synthetic sequences of 7×7 binary images. To address these limitations, this work amends the model, assessing and comparing the effect of each variant of the algorithm. To further explain the limitations, a short description of all the implemented variants is needed.

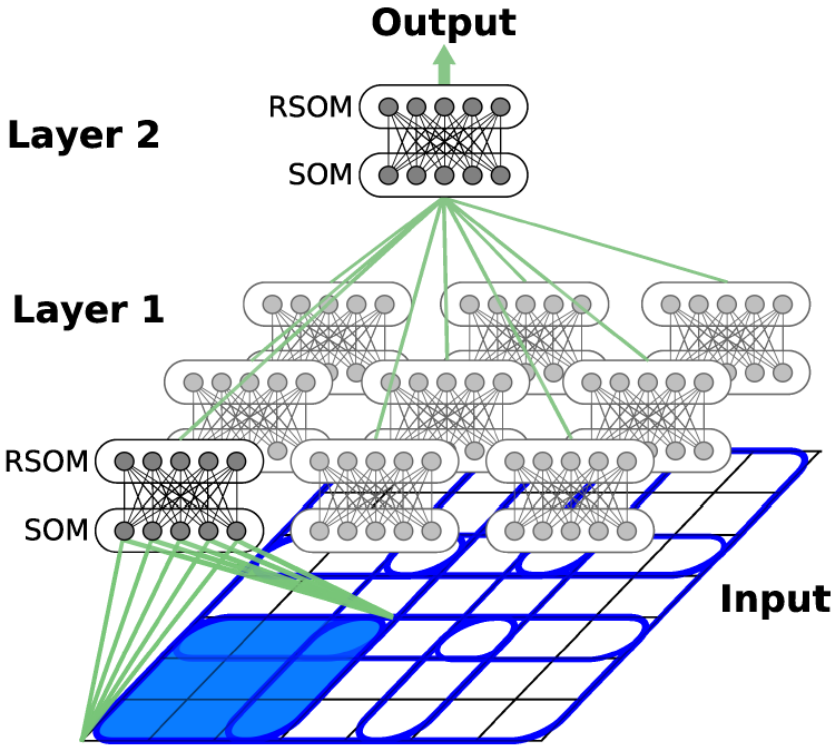


Figure 4.1: HQSOM structure, taken from [66]

4.1 Self-Organizing Maps

The self-organizing map, sometimes referred to as the Kohonen map [70], is perhaps the most popular unsupervised clustering algorithm. It performs a type of dimensionality reduction from the input space dimensions, to the map space V_o (usually two-dimensional). The output of each map is a discrete representation of the training samples input space. The SOM has the unique property of preserving the topology of the input space, meaning that samples that are close to each other in feature space remain close in map space. Maps in the SOM consist of a grid of nodes or elements each representing an input instance by a set of weights w_i for every element i of V_o. At every iteration of the training phase, a distance metric is calculated between all the map weights and the input vector $x(t)$, then a best matching unit (BMU) b is found according to the following equation:

$$||x(t) - w_b|| = min\left(||x(t) - w_i||\right) \tag{4.1}$$

where $|| * ||$ is a distance measure which, in our case, is the Euclidean distance. Then all the weights of a neighborhood of the BMU are shifted towards the input, using the update rule:

$$w_i(t+1) = w_i(t) + \gamma h_{ib}(t)\left(x(t) - w_i(t)\right) \tag{4.2}$$

γ is the learning rate and ranges between 0 and 1, and is often decreased over the course of training. Note that weights are shifted based on the neighborhood function, h_{ib}, which in turn depends on the distance (in map space) between elements i and the BMU b, and it is typically defined as a Gaussian:

$$h_{ib}(t) = \exp\left(\frac{-||I_i - I_b||^2}{\mu_b(t)\sigma^2}\right) \tag{4.3}$$

where I_b and I_i are the indices of the BMU b and element i in map space, $\sigma(t)$ is a neighborhood scaling constant, and $\mu_b(t)$ is the mean square error of comparing the input $x(t)$ to w_b. Using the main square error dynamically adjusts the neighborhood size of the update, in effect, adjusting to new inputs even in the testing phase.

Recurrent Self Organizing Maps

For temporal clustering, the RSOM is used [71] mainly because of its robustness, simplicity, and the elegance of using one algorithm for both spatial and temporal clustering. The input for the RSOM $A(t)$ is obtained from an activation vector defined by the following equation:

$$A(t) = \exp\left(\frac{-||I_i - I_b||^2}{2\rho^2}\right) \tag{4.4}$$

where ρ is the standard deviation of a Gaussian function centered around the BMU index in map space. Lower values of ρ result in a dense representation of the spatial input, while higher values are useful for better generalization, especially for larger training sets. As for the RSOM weights update, the SOM update equation is modified as follows:

$$y_i(t+1) = (1-\alpha)y_i(t) + \alpha\left(A(t) - w_i(t)\right) \tag{4.5}$$

$$w_i(t+1) = w_i(t) + \gamma h_{ib}(t) y_i(t) \tag{4.6}$$

where $y_i(t)$ is considered to be the recursive difference between the input and the previous weights of the map $w_i(t-1)$, and is controlled by the parameter $\alpha, 0 \leq \alpha \leq 1$, specifying the responsiveness of the map to inputs from earlier iterations. When α tends to zero, the RSOM maintains a longer-term memory, while when α is equal to one, the RSOM update is equivalent to that of the SOM, ignoring the temporal aspect.

Parameter-less Self Organizing Maps

In the HQSOM model, as was introduced in [66], a single SOM-RSOM pair has eight parameters to set. There is little theoretical basis on which one can set the values of those parameters. Moreover, the number of parameters increases dramatically in the case of the HQSOM, where there are several layers each consisting of multiple SOM-RSOM pairs. This requirement alone makes the use of HQSOM practically impossible, as tuning all these parameters manually would be extremely time-consuming. We amend the model by employing a parallel implementation of the improved Pa-

rameterless SOM (PLSOM2) [72], which automatically adapts the learning rates and neighborhood functions of the model, effectively reducing the number of parameters to two per SOM-RSOM pair: the responsiveness parameter α, and the generalization parameter ρ. Both parameters are data/problem dependent. The first specifies the RSOM sensitivity to inputs from earlier iterations, and should be set based on the length of the input sequence for a specific class, while the second parameter is directly related to the size of the dataset and the dimensions of the SOM. It controls the SOM output activation vector $A(t)$, and its ability to associate variations in the spatial input between instances of the same class.

The PLSOM2 is an improvement on the original parameterless SOM (PLSOM) [73], in the sense that the PLSOM uses the maximum error encountered during training, to scale the weight update functions, while the PLSOM2 achieves the same scaling but based on the range of the inputs observed so far in the training samples. Thus, the PLSOM2 effectively overcomes the drawbacks of its predecessor, which are: the oversensitivity to extreme outliers, and the dependence on the SOM node weights initial value distribution. PLSOM2 has two stages. First, it calculates the input space size $S(t)$, based on the training samples encountered up to this iteration t. Then, it updates every map node weight vector w_i, but instead of using the learning rate γ and the neighborhood function h_{ib}, PLSOM2 computes a single scaling factor from the previously computed input space size. The input space size is defined as the dataset diameter at time t.

$$S(t) = max_{i,j} \left(||x_i - x_j||^2 \right), \qquad i, j \leq t \tag{4.7}$$

Since x_i is the input at time i, therefore the above equation calculates the maximum distance between all the training samples processed up to t. However, this calculation requires the storage and the diameter calculation of almost the entire training dataset, which in turn is naturally very time and memory consuming. For this reason, in [72] the authors of the PLSOM2 propose an approximation for determining $S(t)$. The algorithm proceeds as follows:

$k \leftarrow n + 1$
$A \leftarrow \emptyset$
$S \leftarrow -1$
for every training sample x **do**
  $s \leftarrow diam(A \cup x)$
  if $s > S$ **then**
    $S \leftarrow s$
    while $size(A) \geq k$ **do**
      $A \leftarrow A - findNearest(x)$
    end while
    $A \leftarrow A \cup x$
  end if
end for

where n is the number of dimension of the input sample vectors, the function $diam(*)$ calculates the diameter of a set, that is the largest distance between any two set members, and the function $findNearest(*)$ computes the distance between an input and a set, returning the member that is nearest to the input. Here we use also the Euclidean distance measures for both functions. The above algorithm works independently of the time t, approximating the calculation of S to a great extent to the value found by equation 4.7, in $O(k(k-1))$ complexity, as reported in [72].

The main power of the PLSOM2 algorithm lies in the map weight update function, that is, unlike the original SOM update, independent of the iteration number. PLSOM2 scales the weight update with the factor $d(t)$ that is defined as:

$$d(t) = min\left(\frac{err(t)}{S}, 1\right) \tag{4.8}$$

Where $err(t)$ is the distance between the input at time t and the BMU or, in other words, the error of the map, and S is the input space size calculated using the approximation algorithm above. Then, a new neighborhood function is defined using $d(t)$ as:

$$\omega(d(t)) = \beta \ \ln\left(1 + d(t)(e-1)\right) \tag{4.9}$$

ln is the natural logarithm, e is the Euler number, and β is the only parameter of the algorithm and is referred to as the neighborhood range. The neighborhood *range* parameter β is an upper bound to the neighborhood *size* parameter σ, and is usually set to the radius of the map. The new neighborhood function $\omega(d(t))$ is then substituted in equation 4.3 forming the following equation:

$$h_{ib}(t) = \exp\left(\frac{-||I_i - I_b||^2}{\omega(d(t))^2}\right) \tag{4.10}$$

As for the learning rate γ, the scaling factor $d(t)$ is used instead, turning equation 4.2 into:

$$w_i(t+1) = w_i(t) + d(t)h_{ib}(t)\left(x(t) - w_i(t)\right) \tag{4.11}$$

The following section provides some details about the parallel implementations of the above mentioned versions of the SOM algorithm.

4.2 Multi-modal Pattern Recognition with HQSOM

Introducing time into the training phase of Self Organizing Maps (SOM) has been addressed by many researchers. The father of self organizing maps, Teuvo Kohonen, introduced a new time-normalized distance operator based on Dynamic Time Warping (DTW), to compute differences between entire sequences of feature vectors with variable length [74]. Training is not performed one feature vector at a time, but the whole sequence is merged into a single matrix, then evaluated against the existing map weights using DTW. It was successfully demonstrated in offline speech recognition of Finnish words. However, it requires batch processing and the prior knowledge of the length of the sequence. Moreover, temporal versions of the Kohonen map were

devised in order to implement some sort of neural feedback, or leaky integrators from previous time step inputs. The Temporal Kohonen Map (TKM) [75], the Recurrent Self Organizing Map (RSOM) [71], and the Recursive Self Organizing Map (Rec-SOM) [76] are the best-known methods. An excellent comparison of those temporal versions can be found in [77]. Finally, the model explained in [78] is very close to the model presented in this work, where the model is made up of a spatial and a temporal SOM on top of each other, with a leaky integrator in between. The authors used it for sequence and sub-sequence classification of musical notes. It improves over previous models, especially the Kangas' model [79], in terms of computational efficiency, also for not requiring a window to be applied over the input sequences. However, the model does not have the potential of being multi-layered or even truly hierarchical, making it biologically implausible.

The HQSOM tries to a great extent to mimic the parallel hierarchical isocortical processing in the brain, using the previously explained pyramid/layered structure. The real power of the model lies in its independence from the modality of the dataset. By subjecting the lower level spatial poolers (the bottom layer of SOMs) to various sensory domains, while the higher levels extract the temporal associations, the model is able to form invariant representations of patterns and objects in spatiotemporal data sequences. This ability, along with the real-time GPU implementation, makes the HQSOM ideal for several applications, especially since it does not require any *a priori* domain knowledge, or data preprocessing. In fact, the model was validated on both raw sensory data and extracted feature vectors, achieving good recognition rates.

Implementation

Apart from the data flow between different layers, which has to be done sequentially, the steps executed by each SOM-RSOM pair are completely independent from the other pairs. Moreover, within a single pair, the BMU search and the maps weight updates can also be easily parallelized for every SOM/RSOM map element. First, we calculate the Euclidean distance between the input vector and the weight vector of

every element, along each dimension, simultaneously in one kernel. Then, the BMU index is found via a minimum parallel reduction kernel, as in equation 4.1. Using the BMU index, the neighborhood function and the map weight update are both performed in parallel by a single kernel. Following this, depending on the learning algorithm employed, if the PLSOM2 method is used we first have to find the input space size $S(t)$, and the scaling factor $d(t)$ to be used in the weight update equations. A parallel version of Algorithm 13 has been devised, where the functions $diam(*)$ and $findNearest(*)$ were each implemented as a CUDA kernel, with as many threads as the input vector dimensions. Either equation 4.6 or equation 4.11 is used for the SOM and RSOM weight updates, where for the SOM the value of the α parameter is always set to 1. Lastly, the activation vector $A(t)$ is created from the BMU of the SOM, using equation 4.4, also in parallel. A CUDA stream is specified for every SOM-RSOM pair, which is useful to execute more than one pair simultaneously if there are enough available resources on the GPU chip.

There are many modes of operation of the HQSOM, one for online learning, where the SOM-RSOM pair node weights are updated in both the training and testing phases. Another mode, which is employed in the case of offline learning (classical SOM equations), in which the maps are updated only in the training phase, while in the testing/-validation step, the activation vector $A(t)$ is formed from the accumulated activation of all the instances of a given sequence, and only moves to a higher level at the end of the whole input validation sequence. The final mode, which is used in the following tests, is based on the PLSOM2 learning algorithm, mainly because the PLSOM2 is relatively slower than the other two learning variants (online and offline). In this mode, only the bottom layer SOMs are trained first, to fully converge to the input dataset, and since we use the parameterless version, the bottom layer can cluster and represent the data without any tuning of the model. After the bottom layer SOMs' convergence, the training is repeated for the higher level SOM-RSOM pairs normally. This effectively reduces the tuning time significantly, providing more robustness to the activation vectors generated directly from the dataset cluster centers (the bottom layer SOMs), and formalizes the training process for different data modalities, where first the model learns the spatial topology of the data, then finds the temporal sequences/patterns of the previously

learned spatial cluster centers. Validation in this mode follows the same procedure as the offline mode testing phase.

Testing and Results

The results of two experiments are presented in this section. In the first, the model is exposed to simulated data of gray scale images, representing a simple moving arrow, while in the second, a public dataset for gesture recognition is presented to the model. For the simulated dataset, the model learns directly from the raw input sequences without any preprocessing; in fact, it is able to successfully classify the patterns even in the presence of noise. On the other hand, the gesture recognition experiment dataset required some preprocessing imposed by the nature of the dataset. This public dataset was acquired from a Microsoft Kinect depth sensor, and is composed of one training instance per gesture, along with multiple validation sequences, each containing one or more test gestures. Hence, the validation set needed to be temporally segmented to extract the input test sequences; moreover, because of the length of the sequences, the training and validation sequences were reduced to have exactly ten frames per gesture. Although this preprocessing is not necessary for the model to learn, it facilitates the manual tuning process of the model parameters, an issue that will be addressed in the following chapter.

Synthetic Sequences

In order to validate the model, a simple simulated time series sequence is employed. The data consists of gray-scale images that are of size 7×7-pixels. There are 3 sequences, each composed of 5 images, representing a horizontally moving arrow, a period of no movement, and a vertically moving arrow, respectively. The arrow is represented by black pixels on a white background, where black has the value of 0 and white 1, the actual dataset is shown in Figure 4.2. Gaussian noise having standard deviation 0.001 was added to all the image pixels while keeping the pixel values within the gray-level $[0, 1]$ range, which means that the random noise values were subtracted

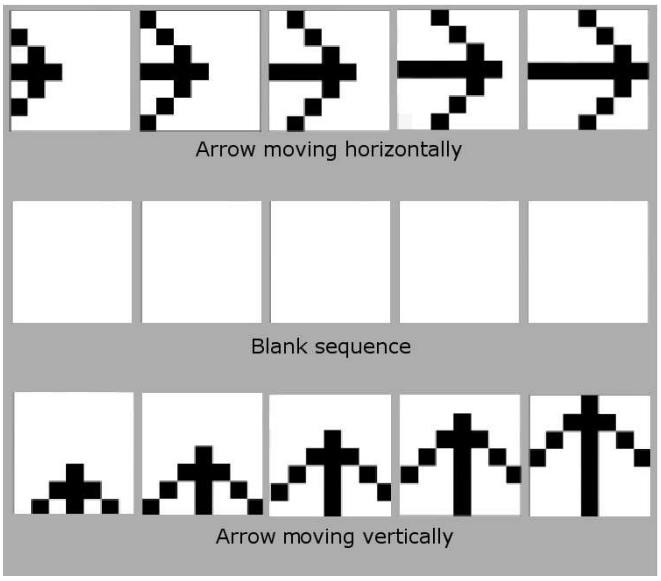


Figure 4.2: Simulated moving arrow sequences

from white pixels and added to the black ones. Each image was introduced to the HQ-SOM for 1000 iterations, with the noise recomputed at every iteration. Since there are only 11 distinct images, 5 (horizontal arrow) + 1 (blank) + 5 (vertical arrow), the SOM size was set to 16 (a 2D grid of 4×4 nodes), while for the RSOM, its size was set to 4 (2×2 grid), as the output classes represent 3 sequences. The rest of the parameters are specified in Table 4.1.

	SOM	RSOM
Learning Algorithm	PLSOM2	PLSOM2
Grid Size	4×4	2×2
Neighborhood Range (β)	2.5	1.75
Memory (α)	1.00	0.45
Activation Sensitivity (ρ)	0.2	-

Table 4.1: Parameters used for the SOM-RSOM pair in the moving arrows classification.

Results

Figure 4.3 shows the SOM-RSOM pair node weights after training. It is clear from the figure the SOM nodes converged correctly to the input samples, where almost every node is directly correlated to a statistical cluster center of the dataset, apart from the five nodes at the top-right of the 2D grid, which are associated with noise or cluster center interpolations. Similarly, RSOM nodes successfully converged to the input sequences, represented as a sequence of cascaded firing of SOM nodes. For instance, looking at the RSOM map, black pixels represent no SOM node activity at the same index as the pixel. On the other hand, higher intensity pixels signify higher occurrence of firing SOM nodes also at the same pixel locations. Thus, through visual inspection, one can infer that node 1 corresponds the the vertically moving arrow sequence. In effect, the HQSOM reduce the dimensionality of the problem from 49 (7×7) dimensions to only 1 single integer output, representing the overall output class RSOM index. It is now enough to examine the output of the model to know the sequence presented at the bottom level of it. However insightful the visual inspection is, a quantification of the results is still necessary. The Probability of Correct Classification P_{CC} is adopted here, which is defined as:

$$P_{CC} = \frac{\sum_{i} max_j \left(v_{ij}\right)}{\sum_{i,j} v_{ij}}$$

Where V is a two dimensional matrix with the correct label given by the row index i, the unsupervised HQSOM output given by the column index j, and each value v_{ij} in the matrix representing the number of times that output j was given for the input corresponding to label i. This finds the simplest mapping from outputs to labels and calculates the likelihood that, using such a mapping, a given input would be matched with its correct label. For 100 independent runs, in each of which the HQSOM was reinitialized, and the noise applied to both training and testing samples, the overall achieved P_{CC} is 99%.

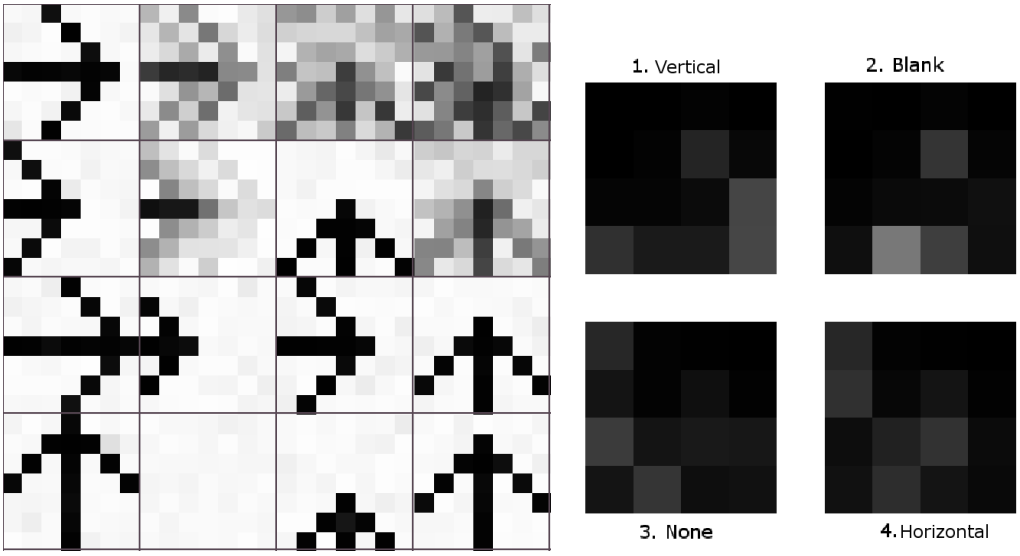


Figure 4.3: (left) 4×4 SOM weights after training from a sample run. (right) 2×2 RSOM weights after training, representing activity/firing sequence of the SOM nodes. Higher pixel intensity signifies SOM nodes that fire together in a given sequence.

Video Sequences

The system was tested on the ChaLearn gesture dataset [80] intended for one-shot learning. This dataset contains video sequences of around 500 batches, each including 100 recorded gestures performed by the same user. We randomly chose a batch for classification, which contains 9 gesture classes, each with only one training example, and 91 test gestures. Two video streams are provided for every gesture sequence, a depth stream acquired by the Kinect sensor along with its synchronized RGB color stream, see Figure 4.4. The main advantages of using the Kinect sensor is that it considerably facilitates user segmentation. To start, we generate motion feature vectors for every frame, by computing the difference between the current frame and the previous one, only from the depth video. Then we automatically choose ten frames per sequence that are representative of the gesture transitions, based on the average total amount of change in the motion feature image. The feature vector/image is then resized to 32×32 pixels. In effect, each feature vector has 1024 dimensions, which is the maximum number of threads that our GPU can currently run in parallel. The exper-

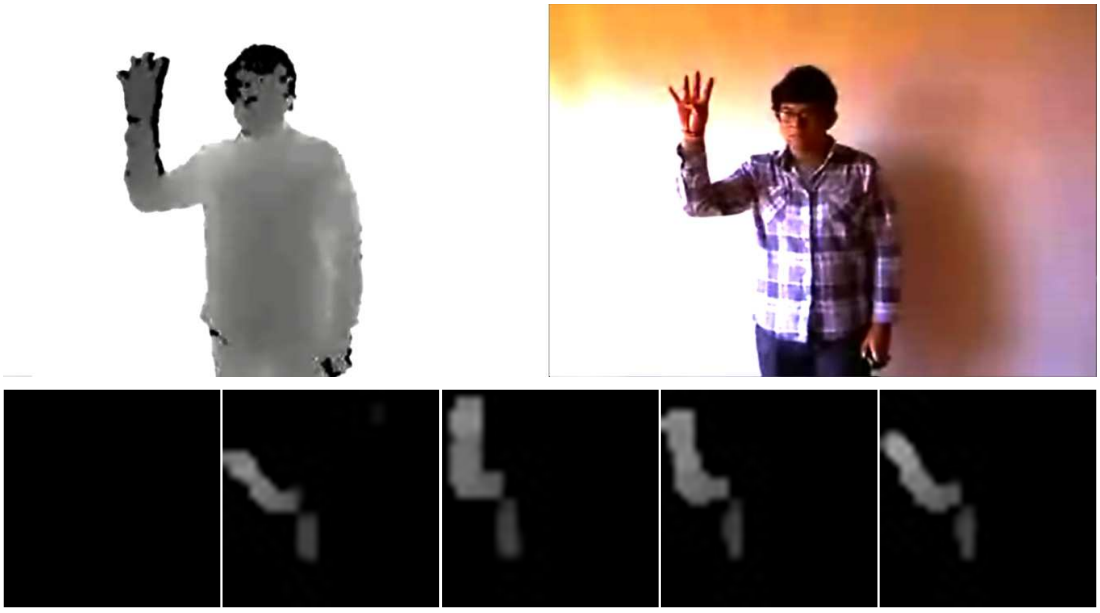


Figure 4.4: Training input example: (top) raw depth and RGB frames, (bottom) five samples of the significant 32x32 rescaled feature frames

iment was run on a 64-bit Intel(R) Core^{TM} i7 CPU running at 2.67GHz using CUDA v. 4.1 on a nVIDIA GeForce GTS450 graphics card with 1GB of DDR memory and compute capability 2.1.

As a proof of concept, we used a single SOM-RSOM pair. The SOM total size was set to 100 (10×10 2D grid), since, having 9 gesture classes each with 10 significant frames, the total number spatial clusters can not be less than 90. Similarly, the RSOM size was set to 25 (5×5 2D grid), which is sufficient to accommodate the number of output classes. The complete set of parameters, obtained through manual tuning, is summarized in Table 4.2. Since the training set is only composed of one sequence for every gesture class, it is not enough to achieve convergence during training. Therefore, the same training set is introduced to the HQSOM for 100 iterations.

	SOM	RSOM
Learning Algorithm	PLSOM2	PLSOM2
Grid Size	10×10	5×5
Neighborhood Range (β)	5.65	4.75
Memory (α)	1.00	0.152
Activation Sensitivity (ρ)	0.8	-

Table 4.2: Parameters used for the SOM-RSOM pair in the ChaLearn gesture recognition system.

Results

After training, the K-Means algorithm is run on the trained RSOM node weights with $k = 9$, to find the RSOM cluster centers corresponding to 9 gesture sequence classes. Then the training set is used once more but as a validation set, in order to assign an RSOM node index to a gesture class ID. This should not be confused with supervised learning, in the sense that the class labels are not used in the learning process, but merely to find correspondence between them and the RSOM output nodes. Since the class labels/annotations are not used in the training phase, the HQSOM can be considered as an unsupervised method.

The SOM and RSOM weights were randomly initialized from a uniform distribution. Thus, the results are dependent on this initial weight distribution. Overall results were obtained from 100 independent runs of the training and validation phases, and are reported in Table 4.3. Moreover, Figure 4.5 shows the SOM-RSOM pair after training from a sample run, along with the overall classification confusion matrix. From the confusion matrix shown, it is apparent that gesture class 4 and 6 are strongly associated. This is due to the fact that those two gestures are performed with the same arm, with the only difference being in the number of fingers shown to the camera. Such a subtle change can not be captured by our motion feature image. Also, the training frames for gesture class 9 were not representative of the gesture pose transitions, hence the performance degradation. It is also important to note that the manual tuning of the model parameters introduces some bias in the results. To obtain the optimal results, an unbiased automatic tuning or model evolution method is crucial.

	Average	StdDev	Worst	Best	Median
Accuracy	69.08%	9.03	41.76%	87.91%	69.23%
Levenshtein Distance	0.3132	0.0865	0.5714	0.1319	0.3187

Table 4.3: Results of ChaLearn gesture classification: average accuracy percentage values, and Levenshtein distances over 100 independent runs.

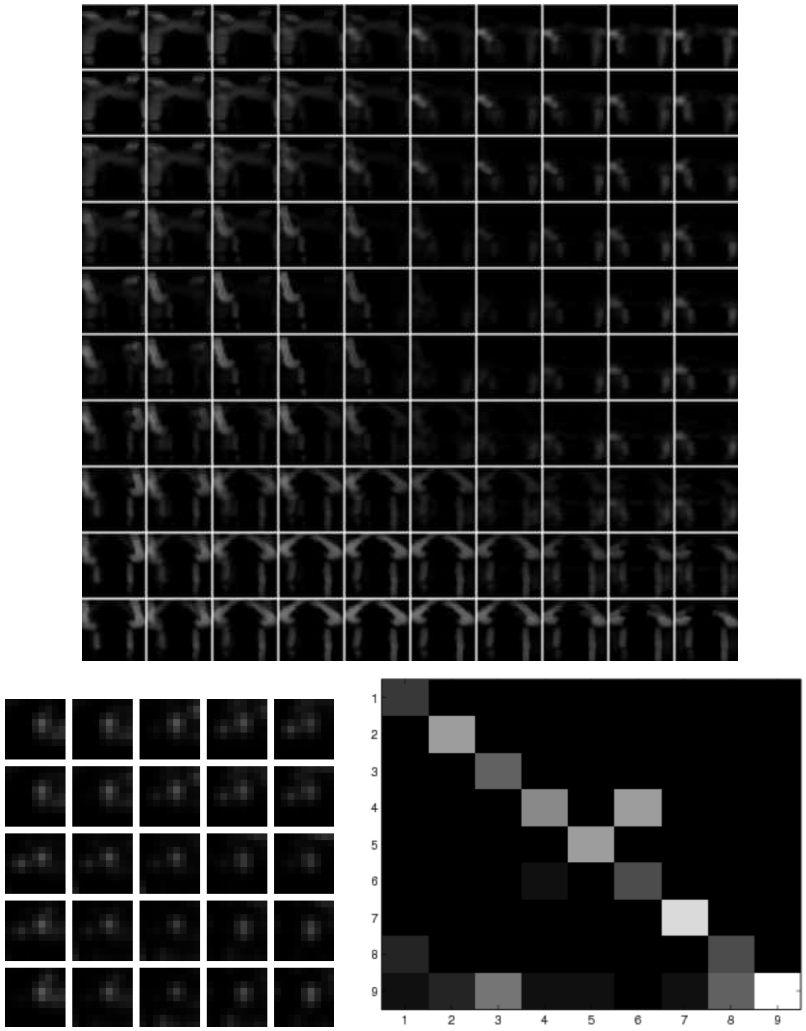


Figure 4.5: (top) 10×10 SOM weights after training from a sample run, showing all the clustered gesture significant frames. (bottom left) 5×5 RSOM weights after training, representing activity/firing sequence of the SOM nodes, higher pixel intensity signify SOM nodes that fire together in a given sequence. (bottom right) A sample classification confusion matrix of the 9 gesture classes

4.3 Final Remarks

The HQSOM model has the ambitious goal of extracting, classifying, and predicting causes and patterns in temporal sequences. The model should be able to achieve these goals in realtime, thanks to the GPU implementation, while maintaining biological equivalence. Currently, we are working on adding feedback connections, to conform with the MPF. We are also investigating activity recognition from Kinect skeleton datasets, where the full power of the method is needed, and the hierarchical approach maps well to the joint hierarchy found in the data. Also, a more extensive testing of the implemented variants is needed, and experiments with different modalities. Finally, a comparison with HTM and the state of the art in temporal classification will shed more light on the advantages of this method.

Chapter 5

Automatic Configuration of the HQSOM

5.1 Parameter Tuning

Very few methods or models claim to be truly parameterless, and even those are usually domain specific and lack the generalization ability of parameterized techniques. Parameters are considered a part of the problem and the solution at the same time. On one hand, an algorithm's parameters can exhibit its flexibility to a multitude of distinct problems and datasets while, on the other hand, finding the best values of those parameters that yield the best algorithm performance is a challenging problem on its own. In most cases, this problem is addressed empirically through trial and error, or what is usually termed manually tuning the algorithm parameters. As it has been mentioned earlier, manual parameter tuning is a time consuming task, and usually does not lead to the optimal values for the parameters under consideration. Manual tuning is performed by systematically changing the value of one parameter while keeping all the other parameter values fixed, until a certain performance criteria crosses a given threshold. This is usually an ad-hoc procedure that in most cases does not guarantee optimality, even more so with an increasing number of parameters, due to the aforementioned

curse of dimensionality issue. Obviously, the more parameters a method has, the more difficult manual tuning becomes. Therefore, there has been a considerable amount of interest and research in automatic tuning of algorithm parameters.

In [52], we used the `irace` package to automatically tune the GPU methods implemented in libCudaOptimize to achieve the best minimization performance over 20 mathematical benchmark functions. The `irace` package implements the iterated racing algorithm, which is an extension of the Iterated F-race procedure, that is based on a statistical approach for selecting the best configuration out of a set of candidate configurations under stochastic evaluation. Its main purpose is to automatically configure optimization algorithms by finding the most appropriate settings given a set of instances of an optimization problem. The scenario usually addressed by `irace` is described as offline configuration [81]. In a preliminary tuning phase, given a set of tuning instances representative of a particular problem, an algorithm configuration is chosen, and in a subsequent production (or testing) phase, the chosen algorithm configuration is used to solve unseen instances of the same problem. The goal is to find, during the tuning phase, an algorithm configuration that minimizes some cost measure over the set of instances that will be seen during the production phase. In general terms, this tuner has been used to solve combinatorial problems [82, 83, 84], but there are also examples of its use in the optimization of metaheuristics for global continuous optimization problems [85].

5.2 HQSOM Tuning via Real Parameter Optimization

The use of optimization methods for automatically tuning algorithm parameters has been studied extensively. In [23], the authors used a Genetic Algorithm (GA) to find the best variants of GA methods given a set of numerical optimization problems, and also given an image registration task. Moreover, [86] used GA's and simulated annealing to automatically tune the scale of the kernel and the regularization parameter of the popular Support Vector Machine (SVM) classification method. Similar to the work in [87], we presented a framework for to estimate the best parameters of PSO

on typical global optimization problems, but using our parallel metaheuristics for this purpose, GPU PSO and DE namely [88].

In the previous chapter it was pointed out that to achieve true data independence in the HQSOM model two main concerns have to be addressed. The first lies in the stimulus sensing layer of the HQSOM, i.e. the SOM part of the first or lowest layer of SOM-RSOM pairs. Since those SOMs are the only part in the model that deal with raw input data, in contrast with the rest of the model which mainly processes SOM and RSOM activations, therefore, tuning their corresponding parameters will have to depend on the input sample sequences value ranges. For this reason, it makes sense to use the PLSOM2 algorithm for the SOM part of the sensing layer learning, effectively eliminating any parameters to tune for this part of the layer, except maybe for the size of the map parameter. As for the rest of the model layers, the PLSOM2 algorithm is both computationally expensive and unfitting. The activation vectors that propagate from lower level layers to the higher ones, also between SOMs and RSOMs of the same layer, vary greatly based on the learning parameters of higher RSOM pairs, the activation density parameter ρ of lower SOMs, and on the RSOM memory parameter α. Those parameters in turn depend on the size of the maps, the number of input sequences, and the sequence length of the input sequences. It is now clear that to adhere more to the biological base of the HQSOM model, and decouple the model's classification ability from the sensory information, those data-dependent parameters have to be automatically estimated or, in other words, evolved based on the fitness of the model, simulating the evolution of our own brains.

Model Formulation

In order to attain the automatic parameter estimation of the HQSOM through function optimization, the optimized model needs to be properly defined first. LibCudaOptimize metaheuristics are used to optimize variants of the HQSOM model, where each candidate solution is equivalent to a complete run of the training and testing phases of the HQSOM classification algorithm. The fitness of a run/candidate solution is calcu-

lated from metrics of the testing phase overall performance. Training phase metrics were not considered in the fitness function, because preliminary tests showed the PL-SOM2 learning algorithm to be more effective than automatically tuned parameters in adapting the SOM/RSOM weights to the training samples.

Fitness Function

Three performance metrics are taken into accaount as terms of the fitness function used in the automatic HQSOM parameter estimation. The first was employed in the previous chapter in the results section, as an alternative to the classification accuracy percentage. The Levenshtein distance [89], sometimes referred to as the edit distance, is a distance measure between two string sequences, and represents the minimum number of single-character edits (insertion, deletion, substitution) required to change one string into the other. The Levenshtein distance $\Delta_{lev}(|A|, |B|)$ between two string sequences $A = a_1 a_2 \cdots a_N$ and $B = b_1 b_2 \cdots b_N$ is mathematically defined as:

$$\Delta_{lev}(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j) = 0 \\ \min \begin{cases} \Delta_{lev}(i-1,j) + 1 \\ \Delta_{lev}(i,j-1) + 1 \\ \Delta_{lev}(i-1,j-1) + [a_i \neq b_i] \end{cases} \end{cases} \tag{5.1}$$

From the above definition it seems that the Levenshtein distance can be computed recursively, where the first element in the minimum corresponds to the cost of deleting a_i, the second to the cost of inserting b_j, and the third to a the cost of replacing a_i with b_j. In practice, the Levenshtein distance is computed using a dynamic programming solution, which involves filling a $(N+1) \times (N+1)$ matrix T. The computation is done according to the base-case rules given by $T[x,0] = T[0,y] = 0$, and the rest of $T[x,y]$ values are filled according to $\Delta_{lev}(i,j)$ defined in equation 5.1.

To assess the HQSOM classification accuracy using the Levenshtein distance, we

consider the output classification label sequence and the ground truth label classification sequence as the two strings A and B to be compared. Here, N is the number of testing samples in the dataset, and each a_i represents the HQSOM output class label for the testing sample i.

The second performance metric used in the fitness evaluation is the total temporal error of the HQSOM, err_{temp}. This metric conveys how well the RSOMs of the model have converged to the training sample sequences, in terms of the whole sequence of lower level SOM firing indexes. It is given by:

$$err_{temp} = \sum_{i=1}^{N} \sum_{j=1}^{L} \sum_{k=1}^{U_j} ||x_i - w_{j,k}^b|| \tag{5.2}$$

where N is the number of testing samples, L the number of HQSOM layers, and U_j the number of SOM-RSOM pairs in layer j. The aggregated error is the Euclidean distance between testing input sequence x_i and the BMU weights $w_{j,k}^b$ of the k^{th} RSOM in the j^{th} layer of the HQSOM. This metric provides an indication of how far or similar the trained weights, represented by the matching RSOM BMUs, are similar to the input sequences.

Finally, the third metric is a simple ratio of the number of unrepresented class labels to the total number of classes R_{class}. This metric is used mainly as a constraint on the fitness function, to prevent the highest classification SOM-RSOM unit from converging to represent multiple classes with one RSOM map element. As described in the previous chapter, the top SOM-RSOM pair is responsible for the overall classification model output, with each RSOM element or cluster center representing a distinct input class. In certain cases, the HQSOM learning algorithm tends to represent more than one input class with a single top RSOM element, sometimes even all the input classes. This behavior is directly affected by the memory parameter α; incorrect values of this parameter makes it difficult for the HQSOM to distinguish between one training sequence and the preceding and following others. Notice that the learning mechanism in this model is unsupervised, meaning no hints were given to the model on when an input training sequence begins or ends. The R_{class} is calculated as $R_{class} = \frac{|\phi|}{K}$, where

ϕ is the set of data classes that have no matching top RSOM elements representing them, and K is the total number of classes in the dataset.

Each of those three fitness metrics provides a different measure: *i.* on the overall classification performance given by Δ_{lev}; *ii.* on the model convergence to the input sequences as described by err_{temp} and controlled by the learning rate (γ) and neighborhood (σ) parameters of the RSOM from equation 4.2 and equation 4.3 respectively, and *iii.* the class representation measured by the ratio R_{class}, which is affected by both the α memory parameter and the SOM activation density ρ shown in equation 4.6 and equation 4.4 respectively. Thus, the fitness function used is composed of the sum of those metrics:

$$f = \Delta_{lev} + err_{temp} + R_{class} \tag{5.3}$$

Testing and Results

Experiments were run on the same Kinect ChaLearn gesture recognition dataset, using the same batch as the experiments from the previous chapter. Testing with the same dataset and application as the manually tuned HQSOM makes it possible to make a fair comparison between the parameters set manually, those set by our parallel meta-heuristics, and also the ones found by a state-of-the-art tuner, `irace`. PSO and DE were employed here for the automatic estimation of the HQSOM parameters, again to compare their optimization performance, but this time on a fitness function that is nondeterministic in its nature. This might create a problem for the search procedure of PSO and DE, as the same candidate solution (HQSOM parameter values) might give better or worse fitness values (classification performance) in a subsequent optimization generation/iteration, depending on the stochastic initialization of the model. For this reason, fitness values will not be always decreasing, in case of function minimization, during the course of the optimization algorithm, as has always happened throughout the previous experiments.

The PSO and DE parameters used in the experiments are summarized in Table 5.1.

DE	PSO
DE/Rand/1 Mutation	$C_1 = 1.49618$
Binomial Crossover	$C_2 = 1.49618$
F = 0.9	$w = 0.729844$
Cr = 0.2	Global Best Topology
Population Size = 20 Number of Generations = 100	

Table 5.1: Parameters of DE/PSO tuners.

Those PSO parameters are set to the 'standard' values suggested in [51] by the creator of PSO, while the parameters for DE are set to the values used in most of our experiments, including the automatic parameter tuning experiment of PSO itself [48, 50, 88]. Of course, another automatic tuner can be used to tune the parameters of the PSO and DE tuners, but then we may fall into an endless loop of automatic tuners. Notice that the population size and number of generations were set to low values, because they greatly affect the computation time of one run of the optimizer. Despite the HQSOM and the tuners' GPU implementations, each particle in the PSO swarm, or member in the DE population, represents a whole run of the HQSOM training and testing stages, which saturates the hardware resources of our single GPU testing machine. For the sake of comparison fairness, the experiment budget for `irace` was also set to 2000 to match the number of HQSOM runs performed by PSO and DE (20 particles/solutions $\times 100$ generations).

Given that the interaction between the particles/elements and the PSO/DE equations themselves are still executed in parallel, in this case, for every generation, each particle fitness has to be computed on the host side, then the fitnesses array transferred to the device side through device global memory, which is a time-consuming operation. The execution time could have been reduced significantly if a multi-GPU system had been available, where the fitness function could have been also executed in parallel, with a single GPU per particle. CUDA version 5.0 introduced the Unified Virtual Address (UVA) ability [90], which will be very useful in this case, where the fitnesses array can be mapped across multi-GPU address spaces, facilitating the memory sharing after every generation.

As stated earlier, the main objective of the optimization is estimating parameters for the higher HQSOM layers RSOM nodes, since the lower layer SOM units are being trained using the PLSOM2 algorithm, virtually requiring no parameters, except for the PLSOM2 neighborhood range parameter, β, which is usually set equal to the radius of the map, or half the map size, as suggested in [72]. Therefore, the SOM parameters of the SOM-RSOM pair used for gesture recognition from the ChaLearn dataset were not adapted during the optimization experiments. The trained SOM weights were loaded from saved values to allow a fair comparison between the manual and automatic tuning of the HQSOM parameters. Table 5.2 summarizes the HQSOM parameters under optimization, along with the allowed ranges for every parameter. Those parameters represent the dimensions of the problem optimized by our GPU optimizers, with the upper and lower bounds of the candidate solution positions per dimension. There are two things to notice from Table 5.2; first, the map size parameter is an integer value, while the implemented metaheuristics are real-valued continuous optimization techniques; for this reason, the rounded integer value of the solution position is set in the model instead of the real valued one. Secondly, the SOM activation density parameter is considered as a parameter of the optimization problem, despite being a parameter of the SOM part of the SOM-RSOM pair. This happens because this parameter does not affect the learning process and the actual weight updates of the SOM, rather, it controls the output vector content passed from a SOM to the corresponding RSOM in a specific pair.

Parameter	Range
RSOM Size	$[3, 32]$
RSOM Neighborhood σ	$[1.5, 32000]$
RSOM Learning Rate γ	$]0.0, 1.0]$
RSOM Memory α	$]0.0, 1.0]$
SOM Activation Density ρ	$]0.0, 1.0]$

Table 5.2: HQSOM parameters to be automatically estimated, and their value ranges.

Tests were run on a 64-bit Intel(R) Core i5 CPU running at 2.5GHz using CUDA v5.0 on a nVidia GeForce GT630M graphics card with 1GB of DDR memory and compute capability 2.1. Performance results are reported in Table 5.3 and Table 5.4,

obtained after 100 independent runs of the HQSOM with the best parameter values found by each optimizer, along with the results of the manually tuned parameters used in the experiments of the previous chapter.

	Average	StdDev	Worst	Best	Median	Holm–Bonferroni
Manual	69.08%	9.03	41.76%	87.91%	69.23%	$< 1.0E-10$
PSO	78.55%	8.02	52.75%	91.21%	81.32%	-
DE	77.53%	7.42	59.34%	91.21%	76.92%	-
`irace`	77.07%	8.87	49.45%	90.11%	79.12%	-

Table 5.3: Gesture classification accuracies of each parameter set, calculated over 100 runs.

	Average	StdDev	Worst	Best	Median	Holm–Bonferroni
Manual	0.3132	0.0865	0.5714	0.1319	0.3187	$< 1.0E-10$
PSO	0.2127	0.0793	0.4725	0.0879	0.1868	-
DE	0.2200	0.0725	0.4066	0.0879	0.2198	-
`irace`	0.2275	0.0869	0.4945	0.0989	0.2088	-

Table 5.4: Levenshtein distances of of each parameter set, calculated over 100 runs.

Two statistical tests were performed to study the existence of pairwise statistical differences among the results of the four methods used for setting the parameters of the HQSOM: pairwise Friedman tests with Dunn-Sidak correction [91] to the p values, and also, the Holm–Bonferroni correction [92] for multiple Wilcoxon signed rank tests. The last column in Table 5.3 and Table 5.4 reports the corrected p-value, setting the significance level to 0.05. The statistical tests found there are statistical differences between the results of the three automatic tuner results and the manual one; on the other hand, the tests found no pairwise significant statistical differences among PSO, DE, `irace`. Figure 5.1 provides another way to visualize the results. Although the results of the automatic tuning using three different methods are very similar, it is clear from the plots that PSO achieves the best results, even surpassing the state-of-the-art in automatic parameter estimation, in terms of both accuracy and distance values, and also in terms of the standard deviation of the obtained results. This, in turn, translates into efficient and robust pattern recognition and classification.

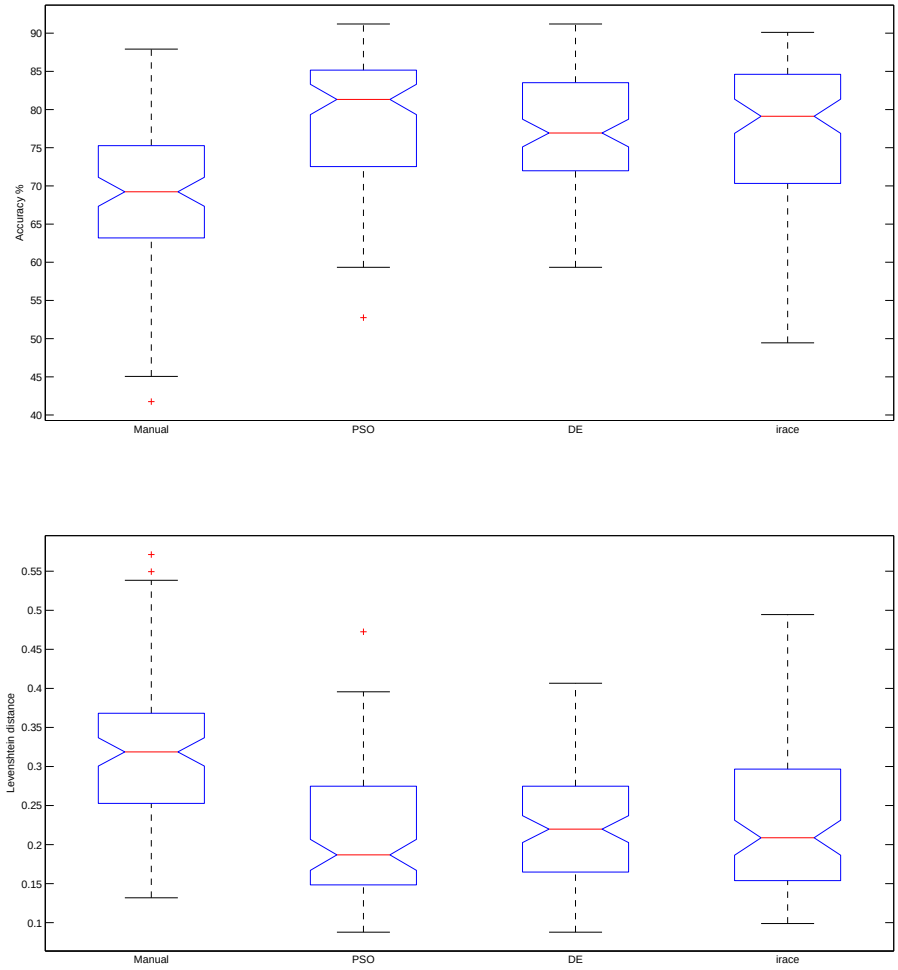


Figure 5.1: Box plots representing the overall performance of each parameter set. (top) Classification accuracy percentages, higher values are better. (bottom) Levenshtein distances between classification output lables and ground truth, lower values are better.

	RSOM Size	RSOM σ	RSOM γ	RSOM α	SOM ρ
PSO	32	32000.0	0.790	0.136	0.689
DE	28	32000.0	0.543	0.117	0.778
`irace`	27	8935.59	0.760	0.130	0.680

Table 5.5: Parameter sets found by both PSO, and `irace`.

Table 5.5 summarizes the automatic parameters estimated by libCudaOptimize, and those found by `irace`. An interesting observation from the table is that the values found by each of the three methods share a lot of similarities. The automatic parameter estimation process provided insight on the effect of the HQSOM parameters that was not obvious from the manual tuning. For instance, all methods set the RSOM map size to near the maximum size allowed (32×32 nodes), with PSO setting it to the maximum, which can be understood as increasing the generalization ability of the time sequence matching units, with RSOM units effectively representing all SOM node index trajectories found in the training data, and compensating for a missing trajectory link in any of the test samples. It can be foreseen that increasing the limits for the RSOM size and neighborhood scaling constant (σ) will result in enhancing the performance of the automatic parameter estimation search. Also worth noting are the values of the SOM activation density parameter ρ, and the RSOM memory parameter α, which are set to almost identical values by PSO and `irace`, meaning the HQSOM has successfully adapted to the input dataset variables, especially the sequence length variable which is affected by the α parameter, all in an unsupervised manner.

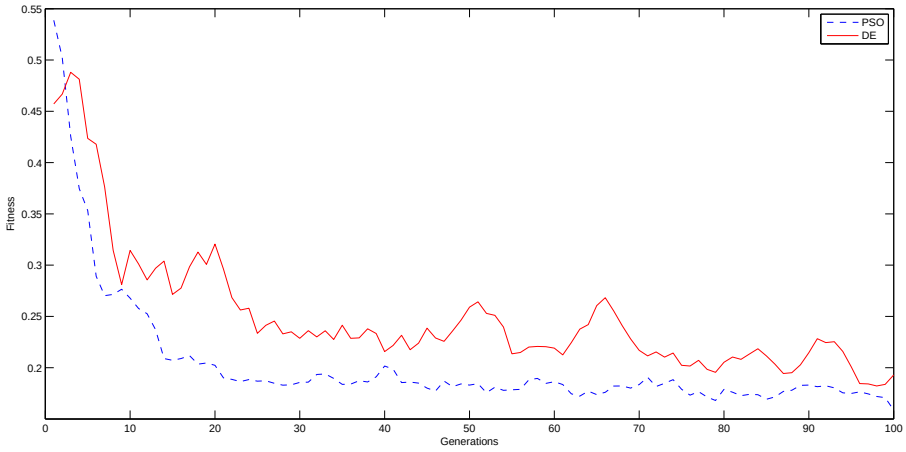


Figure 5.2: Fitness values vs number of generations for PSO and DE in the automatic HQSOM parameter estimation.

Studying the best fitness updates along a metaheuristic course of action, or through its generations, can shed some light on where a specific metaheuristic excels, stagnates,

converges, or prematurely converges. Figure 5.2 shows a plot of the fitness values for the PSO and DE runs. It is obvious that PSO achieves better fitness values than DE throughout its execution course, keeping in mind that the standard parameter values used for PSO are far more well studied than those used in DE. Moreover, PSO appears to have a higher resistance to the nondeterministic nature of the fitness function, which is probably because of the global best topology employed in this experiment, where the swarm is always trying to converge to the best particle position, preserving the best fitness results found so far. It is also noticeable from the plot that both PSO and DE have not yet converged to the global optimum, as the fitness values are still improving up to the last generation. In this case, increasing the number of generations should give the optimizers a chance to converge. Finally, we believe a complete study of the effects of PSO and DE parameters, self or auto-tuning of these parameters, will also result in a better comparison of those optimization methods for the task of automatic parameter estimation.

5.3 Final Remarks

This chapter investigated the use of GPU-based metaheuristics, implemented using libCudaOptimize, to automatically tune the parameters of the HQSOM model, for the ChaLearn gesture recognition application. The parameters were set over a dataset of training and testing video sequence samples of a human subject performing a gesture. The classification/recognition task was formulated as an optimization problem, where the optimizer is trying to minimize an error/fitness function representing the overall performance of our bio-inspired parallel classifier. Since this classifier uses SOMs as its building blocks, the final classification output of the model is nondeterministic, based on the random initialization of the SOM weights. libCudaOptimize proved efficient in solving this optimization problem, without any domain specific (*a priori*) knowledge, apart from the fitness function itself, which is independent from the optimizer implementations. To verify the quality of the results obtained by our tuners, we compared the results they obtained with those obtained using the manu-

ally tuned parameters, and the ones obtained by another state-of-the-art tuning method (`irace`). The results achieved by the algorithm tuned using DE and PSO were generally better than the ones acquired manually, and slightly better than those achieved by `irace`. Keeping in view that `irace` was designed specifically to statistically compare different parameter configurations of stochastic and evolutionary algorithms, repeating a single configuration multiple times to verify the effects of its parameter values, while metaheuristics are general optimization methods that through our implementation proved also to be effective in the automatic parameter estimation of a stochastic classification method.

Based on the statistical tests, we can conclude that all the three metrics in the fitness function, defined in equation 5.3 and used by all the automatic tuners, have a positive effect on guiding the search process to the global optimum of the complicated fitness landscape of the HQSOM parameter values search space. It is important to note that the optimization performance is upper-bounded by the number of particles and generations, which, as was stated earlier, were set to low values because of the computational complexity incurred by increasing those numbers. Moreover, the overall gesture classification accuracy still suffers from the problems arising from resizing the feature vectors to 1024 dimensions, to be processed by 1024 threads simultaneously, see Figure 4.4. It is clear that all the algorithm limitations are hardware based, and are expected to be overcome with the newer, more powerful GPUs, and GPU clusters.

Chapter 6

Further Work

All the methods that were demonstrated in this thesis have been implemented using the nVIDIA CUDA framework, thus, other interesting developments may be offered by the availability of OpenCL, which will allow owners of different GPUs (as well as multi-core CPUs, which are also supported) than nVIDIA's to implement parallel algorithms on their own computing architectures. The availability of shared code which allows for optimized code parallelization even on more traditional multi-core CPUs will make the comparison between GPU-based and multi-core CPUs easier (and, possibly, fairer) besides allowing for a possible optimized hybrid use of computing resources in modern computers.

Regarding libCudaOptimize future developments, several aspects can be improved or extended. Our next efforts will mainly be concerned about: the realization of some visualization and statistical tools in order to help behavioral and performance analysis of metaheuristics; more support for multiple solution sets, like allowing different sets to have independent termination criteria; the possibility to evolve solutions of data types other than floats; and the parallel implementation of other well-known optimization methods like Genetic Algorithms, Evolution Strategies or Evolutionary Programming as well as further expansions of the methods already present. Also, the library needs extensive documentation and a user manual. Finally, it needs support for combinatorial optimization problems, and a way to represent constrained optimization

problems.

As for the HQSOM, currently we are testing a full hierarchy of SOM-RSOM pairs on public datasets like the Microsoft Research Cambridge-12 (MSRC-12) gesture dataset [93], which consists of sequences of human skeletal body part movements (represented as body part locations) and the associated meaning that needs to be recognized by the model. For this dataset, we have a HQSOM that is composed of three layers: the bottom sensory layer has five SOM-RSOM cells, each processing different body part information, namely, right arm, right leg, left arm, left leg, and torso 3D skeletal joint positions. The mid layer consists of three cells, each grouping the features and sequences extracted by the lower layer into right, left, and middle body movements. Lastly, the top layer consists of a single SOM-RSOM cell, that is responsible for forming invariant representations of the whole body movements through time, again based on the features clustered by the second layer. The Australian Sign Language signs (AUSL) Data Set [94] is another dataset against which the HQSOM is being verified. The AUSL dataset consists of sample of Auslan (Australian Sign Language) signs. 27 examples of each of 95 Auslan signs were captured from a native signer using high-quality position tracker gloves. The HQSOM employed for this dataset, has two layers: the sensory layer has two cells, one for the left hand and the other for the right hand, each processing an eleven dimension feature vector of the following format: (3D hand position, 3 axial rotation angles, and 5 finger bend values). Similarly, the top layer is composed of one SOM-RSOM cell, whose input is the RSOM element index output of the lower layer (left hand cell + right hand cell), and its output the top RSOM sequence index representing an Auslan sign class. Preliminary results of HQSOM training on both those datasets exhibits the potential of using such a model to detect and recognize patterns in spatio-temporal data, in an unsupervised way, as the model's building block, the SOM, is able to preserve and make use of the data's spatial topology, while, on the other hand, the RSOM is designed to cluster the detected spatial patterns through time. Ultimately, we would like to release the source code of the HQSOM model to the public, as an open-source library, as we did with libCudaOptimize. However, this will involve extensive documentation and refactoring of the source code.

Finally, as was mentioned in the previous chapter, the use of libCudaOptimize to automatically tune the HQSOM parameters suffers from some limitations, mainly pertaining to the computational complexity resulting from the fitness function being a complete run of another GPU-based method. Therefore, one can not make use of the fine-grained parallelization, because of GPU resource limitations. This hardware constraints can be addressed by using a multi-GPU system, or a large CPU cluster, which would make it possible to run the parameter estimation optimization with larger populations for more iterations, achieving better results, and even allowing the automatic tuning of the libCudaOptimize parameters itself. Another alternative would be to calculate the fitness of a HQSOM parameter set 'online', or during the execution time of the training phase, using some sort of performance metric, most likely computed from the mean square error between the inputs and the BMUs of the corresponding SOM-RSOM pair. The goal here is to decouple the model from the data being processed, which can be enhanced by also estimating the parameters of the lowest level SOM (the sensory layer), including the size of the receptive fields of this layer's cells, effectively removing the need for the parameter-less training algorithm (PLSOM2), which is also another time consuming stage. Once the experiments with the MSRC-12 and the AUSL datasets are complete, it would be certainly interesting to optimize a full HQSOM with many layers, verifying whether the estimated parameters can detect and adapt to the spatial boundaries found in the input signals, forming appropriate receptive fields, akin to the early development of neuronal layers in the primary visual cortex.

Chapter 7

Summary and Conclusions

Bio-inspired models and methods tend to be intrinsically parallel. This thesis investigated the benefits of such parallel models in terms of efficiency and accuracy. It started by the implementation of a GPU version of the Asynchronous Particle Swarm Optimization (PSO), using the nVIDIA CUDA platform on consumer-level Graphics Processing Units (GPU), then comparing it in terms of execution time and accuracy to the existing parallel synchronous PSO algorithm. Also, it detailed the implementation of a parallel version of the Differential Evolution (DE) and Scatter Search (SS) optimization algorithms, and their integration into a common framework that enabled the comparison between DE, PSO, SS, and any other population based optimization method. The comparison was in terms of the fitness values achieved by the GPU methods over a benchmark of 20 popular mathematical functions, specifying execution time as a termination criterion, effectively assessing the parallelization potential of each optimization method.

While implementing those parallel techniques, similarities between the designs of the population based optimization algorithms appeared. Realizing this, we managed to abstract the core of the GPU algorithms to create an open source library (libCudaOptimize), to be used by the community or anyone with a continuous parameter optimization problem. The library is a C++ API that handles all the GPU thread and memory allocation, the parallel optimization method, and the statistics for comparing

one method to the other. Users have only to provide their own optimization problem (fitness function). Using the GPU-based PSO and DE we managed to address real-world problems like road sign detection and classification, and human body pose estimation. Also, the libCudaOptimize library enabled the researchers in the IBIS lab to easily compare two popular metaheuristics (PSO and DE) in the context of object detection in 2D images and videos. We formulated object detection as a continuous optimization problem, where the parallel optimization method generates candidates of a deformable model, specifying the object to be detected, while the fitness of these candidates is the degree of overlap between the model and the input media (images/video). The experiments for this comparison were run for two real world applications. First, the Hippocampus localization in histological images, and second, the human body pose estimation from multi-view video sequences.

Afterwards, the thesis focused on Kinect gesture recognition using a parallel model of the neocortex, namely the Hierarchical Quilted Self Organizing Maps (HQSOM). Since all neural based models are also inherently parallel, a parallel version of this neocortex model was implemented on the GPU, and expanded by using a new kind of Self Organizing Maps (SOM), called the Parameter-less Self Organizing Map (PLSOM2). This choice was motivated by the HQSOM requirement of many levels of interacting SOMs, arranged in a tree structure. Therefore, decreasing the number of model parameters is paramount to applying the model to real-world classification problems. The model was verified on the Microsoft ChaLearn Kinect gesture dataset, achieving good classification results. Lastly, merging the two main parts of our research, the HQSOM model parameters were set to be optimized by libCudaOptimize, more specifically by PSO, essentially achieving automatic adaptation of the model parameters based on overall classification performance. This, in essence, prepares the model to find, predict, and classify patterns in any temporal signal from different modalities.

In summation, we proved the parallel approach to implementing computational models is able to leverage the increasing processing power of multi-core architectures to solve real-world problems, while injecting the specific domain intelligence through the distributive collaboration of nature-inspired model elements. Our own brains have

evolved to tune this very same parallel approach. Therefore, simulating this process, through evolutionary computation and swarm intelligence, along with the parallel processing capabilities of modern supercomputers, will eventually lead to general methods that can adapt to the different data requirements of current scientific questions.

Bibliography

[1] D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[2] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Proc. IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[3] Riccardo Poli. Analysis of the publications on the applications of Particle Swarm Optimisation. *J. Artificial Evolution and Applications*, pages 1–10, 2008.

[4] Alec Banks, Jonathan Vincent, and Chukwudi Anyakoha. A review of Particle Swarm Optimization. Part I: background and development. *Natural Computing*, 6:467–484, 2007.

[5] Luca Mussi, Fabio Daolio, and Stefano Cagnoni. Evaluation of parallel particle swarm optimization algorithms within the CUDA™ architecture. *Information Sciences*, 181(20):4642–4657, 2011.

[6] Luca Mussi, Youssef S. G. Nashed, and Stefano Cagnoni. GPU-based asynchronous particle swarm optimization. In *Proc. 13th annual conference on Genetic and Evolutionary Computation, GECCO '11*, pages 1555–1562. ACM, 2011.

[7] R. Storn and K. Price. Differential Evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces. Technical report, International Computer Science Institute, 1995.

[8] J. Vesterstrom and R. Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Proc. IEEE Congress on Evolutionary Computation*, pages 1980–1987, 2004.

[9] S. Das and P.N. Suganthan. Differential Evolution: A Survey of the State-of-the-Art. *IEEE Transactions on Evolutionary Computation*, 15(1):4–31, 2011.

[10] Ferrante Neri and Ville Tirronen. Recent advances in differential evolution: a survey and experimental analysis. *Artif. Intell. Rev.*, 33:61–106, 2010.

[11] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[12] Rafael Marti, Manuel Laguna, and Fred Glover. Principles of scatter search. *European Journal of Operational Research*, 169(2):359–372, 2006.

[13] Francisco J. Wets and Roger J.B. Solis. Minimization by random search techniques. *Mathematics of Operations Research*, 6(1):19–30, 1981.

[14] Tomaso Poggio and Emilio Bizzi. Generalization in vision and motor control. *Nature*, 431(7010):768–774, 2004.

[15] N.J. Shah, J.C. Marshall, O. Zafiris, A. Schwab, K. Zilles, H.J. Markowitsch, and G.R. Fink. The neural correlates of person familiarity. *Brain*, 124(4):804–815, 2001.

[16] J. Hawkins and S. Blakeslee. *On intelligence*. Owl Books, 2005.

[17] Dileep George and Jeff Hawkins. A hierarchical bayesian model of invariant pattern recognition in the visual cortex. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 3, pages 1812–1817. IEEE, 2005.

[18] Saulius J Garalevicius. Memory-prediction framework for pattern recognition: Performance and suitability of the bayesian model of visual cortex. In *FLAIRS Conference*, pages 92–97, 2007.

[19] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26:80–113, 2007.

[20] G. Hager, T. Zeiser, and G. Wellein. Data access optimizations for highly threaded multi-core cpus with multiple memory controllers. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008.*, pages 1–7, april 2008.

[21] nVIDIA Corporation. *nVIDIA CUDA Programming Guide v. 4.0*, May 2011.

[22] nVIDIA Corporation. *CUDA C Best Practices Guide v. 4.0*, 2011.

[23] J Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern.*, 16(1):122–128, 1986.

[24] Sean Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[25] Nikolaus Hansen, Steffen Finck, Raymond Ros, and Anne Auger. Real-Parameter Black-Box Optimization Benchmarking 2009: Noiseless Functions Definitions. Research report, INRIA, 2009.

[26] Nikolaus Hansen, Raymond Ros, Nikolas Mauny, Marc Schoenauer, and Anne Auger. PSO facing non-separable and ill-conditioned problems. Research Report RR-6447, INRIA, 2008.

[27] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George. Parallel global optimization with the particle swarm algorithm. *Journal of numerical methods in engineering*, 61:2296–2315, 2003.

[28] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. *journal of aerospace computing, information, and communication*, 2005.

[29] J. Li, D. Wan, Z. Chi, and X. Hu. An efficient fine-grained parallel particle swarm optimization method based on GPU-acceleration. *International Journal of Innovative Computing, Information and Control*, 3(6 B):1707–1714, 2007.

[30] L. de P. Veronese and R.A. Krohling. Swarm's flight: Accelerating the particles using C-CUDA. In *IEEE Congress on Evolutionary Computation (CEC), 2009*, pages 3264–3270, May 2009.

[31] Y. Zhou and Y. Tan. Particle swarm optimization with triggered mutation and its implementation based on GPU. In *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference (GECCO), 2010*, pages 1007–1014, 2010.

[32] Weihang Zhu. Massively parallel differential evolution–pattern search optimization with graphics hardware acceleration: an investigation on bound constrained optimization problems. *Journal of Global Optimization*, 50(3):417–437, 2011.

[33] J. St.Charles, T.E. Potok, R. Patton, and X. Cui. Flocking-based document clustering on the graphics processing unit. *Studies in Computational Intelligence*, 129:27–37, 2008.

[34] Weihang Zhu and Yaohang Li. GPU-accelerated differential evolutionary markov chain monte carlo method for multi-objective optimization over continuous space. In *Proceeding of the 2nd workshop on Bio-inspired algorithms for distributed systems*, BADS '10, pages 1–8, New York, NY, USA, 2010. ACM.

[35] B-I. Koh, A. D. George, R. T. Haftka, and B. J. Fregly. Parallel asynchronous particle swarm optimization. *International Journal for Numerical Methods in Engineering*, 67:578–595, 2006.

[36] L. Dioşan and M. Oltean. Evolving the structure of the particle swarm optimization algorithms. In *European Conference on Evolutionary Computation in Combinatorial Optimization, EvoCOP'06*, pages 25–36. Springer Verlag, 2006.

[37] L. Dioşan and M. Oltean. What else is evolution of PSO telling us? *Journal of Artificial Evolution and Applications*, 1:1–12, 2008.

[38] nVIDIA Corporation. *CUDA Toolkit 4.0 CURAND Guide*, 2011.

[39] L.P. de Veronese and R.A. Krohling. Differential evolution algorithm on the GPU with C-CUDA. In *Proc. IEEE Congress on Evolutionary Computation*, pages 1–7, 2010.

[40] Pavel Krömer, Václav Snášel, Jan Platoš, and Ajith Abraham. Many-threaded implementation of differential evolution for the CUDA platform. In *Proc. 13th annual conference on Genetic and Evolutionary Computation, GECCO '11*, pages 1595–1602. ACM, 2011.

[41] Pavel Krömer, Václav Snášel, Jan Platoš, and Ajith Abraham. A comparison of many-threaded differential evolution and genetic algorithms on CUDA. In *Nature and Biologically Inspired Computing (NaBIC), 2011 Third World Congress on*, pages 509 –514, 2011.

[42] Fred Glover. A template for scatter search and path relinking. In *Artificial evolution*, pages 1–51. Springer, 1998.

[43] Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and cybernetics*, 39(3):653–684, 2000.

[44] Fred Glover. Tabu search - Part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[45] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval-schemata. In Darrell L. Whitley, editor, *Foundation of Genetic Algorithms 2*, pages 187–202, 1993.

[46] Abraham Duarte, Rafael Martí, Fred Glover, and Francisco Gortázar. Hybrid scatter tabu search for unconstrained global optimization. *Annals of Operations Research*, 183(1):95–123, 2011.

[47] Youssef S. G. Nashed, Roberto Ugolotti, Pablo Mesejo, and Stefano Cagnoni. libcudaoptimize: an open source library of GPU-based metaheuristics. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion*, pages 117–124. ACM, 2012.

[48] Roberto Ugolotti, Youssef S. G. Nashed, Pablo Mesejo, Špela Ivekovič, Luca Mussi, and Stefano Cagnoni. Particle swarm optimization and differential evolution for model-based object detection. *Applied Soft Computing*, 2012.

[49] Luca Mussi, Spela Ivekovic, Youssef S. G. Nashed, and Stefano Cagnoni. Multi-view human body pose estimation with CUDA-PSO. *International Journal of Adaptive, Resilient and Autonomic Systems (IJARAS)*, 3(4):51–65, 2012.

[50] Roberto Ugolotti, Youssef S. G. Nashed, and Stefano Cagnoni. Real-time GPU based road sign detection and classification. In *Parallel Problem Solving from Nature-PPSN XII*, pages 153–162. Springer Berlin Heidelberg, 2012.

[51] J. Kennedy and M. Clerc, 2 2006. http://www.particleswarm.info/Standard_PSO_2006.c.

[52] Youssef S. G. Nashed, Pablo Mesejo, Roberto Ugolotti, Jérémie Dubois-Lacoste, and Stefano Cagnoni. A comparative study of three GPU-based metaheuristics. In *Parallel Problem Solving from Nature-PPSN XII*, pages 398–407. Springer Berlin Heidelberg, 2012.

[53] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Thomas Stützle, and Mauro Birattari. The irace package, iterated race for automatic algorithm configuration. Technical Report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[54] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm Intelligence*, 1(1):33–57, 2007.

[55] P N Suganthan, N Hansen, J J Liang, K Deb, Y Chen, A Auger, and S Tiwari. Problem definitions and evaluation criteria for the CEC 2005 special session on real-parameter optimization. *Natural Computing*, pages 1–50, 2005.

[56] Jan Bandouch, Florian Engstler, and Michael Beetz. Evaluation of hierarchical sampling strategies in 3D human pose estimation. In *Proceedings of the 19th British Machine Vision Conference (BMVC)*, 2008.

[57] Jonathan Deutscher and Ian Reid. Articulated body motion capture by stochastic search. *Int. J. Comput. Vision*, 61:185–205, February 2005.

[58] John Maccormick and Michael Isard. Partitioned sampling, articulated objects, and interface-quality hand tracking, 2000.

[59] Fabrice Caillette, Aphrodite Galata, and Toby Howard. Real-time 3-d human body tracking using learnt models of behaviour. *Comput. Vis. Image Underst.*, 109:112–125, February 2008.

[60] Raquel Urtasun, David J. Fleet, Aaron Hertzmann, and Pascal Fua. Priors for people tracking from small training sets. In *Proceedings of the Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, pages 403–410, Washington, DC, USA, 2005. IEEE Computer Society.

[61] Juergen Gall, Bodo Rosenhahn, Thomas Brox, and Hans-Peter Seidel. Optimization and filtering for human motion capture. *Int. J. Comput. Vision*, 87:75–92, March 2010.

[62] Špela Ivekovič, Emanuele Trucco, and Yvan R. Petillot. Human body pose estimation with particle swarm optimisation. *Evol. Comput.*, 16:509–528, December 2008.

[63] Vijay John, Emanuele Trucco, and Špela Ivekovič. Markerless human articulated tracking using hierarchical particle swarm optimisation. *Image Vision Comput.*, 28:1530–1547, November 2010.

[64] J. Warren and S. Schaefer. A factored approach to subdivision surfaces. *Computer Graphics and Applications*, 24(3):74–81, 2004.

[65] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, pages 80–83, 1945.

[66] J.W. Miller and P.H. Lommel. Biomimetic sensory abstraction using hierarchical quilted self-organizing maps. In *Proceedings-SPIE the International Society for Optical Engineering*, 2006.

[67] K. Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[68] S. Behnke and R. Rojas. Neural abstraction pyramid: A hierarchical image understanding architecture. In *IEEE World Congress on Computational Intelligence*, volume 2, pages 820–825. IEEE, 1998.

[69] S.M. Stringer and E.T. Rolls. Invariant object recognition in the visual system with novel views of 3d objects. *Neural Computation*, 14(11):2585–2596, 2002.

[70] T. Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[71] M. Varstal, J. Millán, and J. Heikkonen. A recurrent self-organizing map for temporal sequence processing. *Artificial Neural Networks ICANN'97*, pages 421–426, 1997.

[72] E. Berglund. Improved plsom algorithm. *Applied Intelligence*, 32(1):122–130, 2010.

[73] Erik Berglund and Joaquin Sitte. The parameterless self-organizing map algorithm. *Neural Networks, IEEE Transactions on*, 17(2):305–316, 2006.

[74] P. Somervuo and T. Kohonen. Self-organizing maps and learning vector quantization for feature sequences. *Neural Processing Letters*, 10(2):151–159, 1999.

[75] G.J. Chappell and J.G. Taylor. The temporal kohønen map. *Neural Networks*, 6(3):441–445, 1993.

[76] M. Strickert, B. Hammer, and S. Blohm. Unsupervised recursive sequence processing. *Neurocomputing*, 63:69–97, 2005.

[77] M. Varsta, J. Heikkonen, J. Lampinen, and J.D.R. Millán. Temporal kohonen map and the recurrent self-organizing map: Analytical and experimental comparison. *Neural processing letters*, 13(3):237–251, 2001.

[78] O.A.S. Carpinteiro. A hierarchical self-organizing map model for sequence recognition. *Neural Processing Letters*, 9(3):209–220, 1999.

[79] J. Kangas. *On the analysis of pattern sequences by self-organizing maps*. PhD thesis, J. Kangas, 1994.

[80] ChaLearn gesture dataset (CGD 2011), ChaLearn, California, 2011.

[81] Mauro Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*. Springer, 2nd edition, 2009.

[82] Jérémie Dubois-Lacoste, Manuel López-Ibáñez, and Thomas Stützle. A hybrid TP+PLS algorithm for bi-objective flow-shop scheduling problems. *Computers & Operations Research*, 38(8):1219–1236, 2011.

[83] Jérémie Dubois-Lacoste, Manuel López-Ibáñez, and Thomas Stützle. Improving the anytime behavior of two-phase local search. *Annals of Mathematics and Artificial Intelligence*, 61(2):125–154, 2011.

[84] Manuel López-Ibáñez and Thomas Stützle. Automatic configuration of multi-objective ACO algorithms. In *Proc. of the 7th international conference on Swarm Intelligence*, ANTS'10, pages 95–106, 2010.

[85] Marco Antonio Montes de Oca, Dogan Aydin, and Thomas Stützle. An incremental particle swarm for large-scale continuous optimization problems: an example of tuning-in-the-loop (re)design of optimization algorithms. *Soft Computing*, 15(11):2233–2255, 2011.

[86] F Imbault and K Lebart. A stochastic optimization approach for parameter tuning of support vector machines. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 4, pages 597–600. IEEE, 2004.

[87] Nobuhiro Iwasaki, Keiichiro Yasuda, and Genki Ueno. Dynamic parameter tuning of particle swarm optimization. *IEEJ Transactions on Electrical and Electronic Engineering*, 1(4):353–363, 2006.

[88] Roberto Ugolotti, Youssef S. G. Nashed, Pablo Mesejo, and Stefano Cagnoni. Algorithm configuration using gpu-based metaheuristics. In *Proceeding of the fifteenth annual conference companion on Genetic and evolutionary computation conference companion*, pages 221–222. ACM, 2013.

[89] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[90] nVIDIA Corporation. *nVIDIA CUDA Programming Guide v. 5.0*, July 2012.

[91] O. J. Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 56:52–64, 1961.

[92] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[93] Simon Fothergill, Helena M. Mentis, Pushmeet Kohli, and Sebastian Nowozin. Instructing people for training gestural interactive systems. In Joseph A. Konstan, Ed H. Chi, and Kristina Höök, editors, *CHI*, pages 1737–1746. ACM, 2012.

[94] Mohammed Waleed Kadous. *Temporal classification: Extending the classification paradigm to multivariate time series*. PhD thesis, The University of New South Wales, 2002.