An improved CUDA-based hybrid metaheuristic for fast controller of an evolutionary robot

Nour El-Houda Benalia*, NourEddine Djedi, Salim Bitam and Nesrine Ouannes

Department of Computer Science,
LESIA Laboratory,
University of Biskra, Algeria
Email: n.benalia@univ-biskra.dz
Email: n.djedi@univ-biskra.dz
Email: salim.bitam@laposte.net
Email: nesrineouannes@gmail.com
*Corresponding author

Yves Duthen

Vortex Team,
IRIT Laboratory,
Toulouse, France
Email: Yves.Duthen@irit.fr

Abstract: This paper proposes a novel parallel hybrid training approach to conceive an evolutionary robot. The proposed design aims to provide efficient behaviours to perform its tasks in a complex area such as walking toward a hidden destination. Embedded in robot brain, this training and evolution combination is typically accomplished by evolving considerable recurrent neural networks (RNNs) using an evolutionary strategy (ES). The effectiveness of this proposal is improved by employing CUDA technology that executes the evolutionary process of RNNs in a parallel way. The modifications applied are indicating to meet CUDA requirements in terms of CPU/GPU cooperation and memory management. Using a set of experiments performed by GPGPU-based physical simulator named open dynamics engine (ODE) and CUDA-based evolution, the effectiveness of the proposed parallel evolutionary training technique was validated for real movements of humanoid robots. This validation showed a promising speed-up, since this field requires very high powerful computational resources.

Keywords: artificial life; robotics; parallel evolutionary algorithms; recurrent neural network; RNN; GPU.

Reference to this paper should be made as follows: Benalia, N.E-H., Djedi, N.E., Bitam, S., Ouannes, N. and Duthen, Y. (xxxx) 'An improved CUDA-based hybrid metaheuristic for fast controller of an evolutionary robot', *Int. J. Embedded Systems*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Nour El-Houda Benalia received her Master degree in Image and Artificial Life from UMK University, Biskra, Algeria in 2010. She is currently a PhD candidate in Computer Science at the LESIA Laboratory at the same university. Her current research interests include GPU computing, high performance technology, bioinspired techniques and artificial life.

NourEddine Djedi is a Professor of Computer Science and conducts research in artificial life and natural phenomenon simulation team of the LESIA Laboratory at the University of Biskra. He received his PhD in Computer Science in 1991 at the University Paul Sabatier (Toulouse), and worked successively in several fields such as image synthesis, bio-inspired techniques and artificial life.

Salim Bitam is an Associate Professor of Computer Science, a senior member of LESIA Laboratory at University of Biskra (Algeria), and an associate member of LiSSi Laboratory at University of Paris-Est (France). He received his PhD from University of Biskra and a Doctorate of Sciences (Habilitation) Diploma from Higher School of Computer Science – ESI (Algiers, Algeria). His main research interests are bio-inspired methods, VANETs and cloud computing. He has to his credit more than 30 publications in journals, books and conferences. He serves as an editorial member and a reviewer of several journals such as IEEE, Elsevier, Wiley and Springer.

Nesrine Ouannes received her Engineering degree, Magister Diploma in Computer Graphics and Artficielle, and prepares currently a PhD in Artificial Life and Ecosystems at Biskra University in Algeria. Her current research interests include the development of artificial creatures behaviours in different levels of simulation and ecosystems.

Yves Duthen is a Professor of Computer Science and conducts research in artificial life and virtual reality team VORTEX of the Institute for Research in Computer Science of Toulouse (IRIT). He received his PhD in Computer Science in 1983 then HDR in 1993, he worked successively in several fields such image synthesis, parallel machines, behavioural simulation and artificial creatures.

This paper is a revised and expanded version of a paper entitled 'An improved CUDA-based hybrid metaheuristic for fast controller of an evolutionary robot' presented at IT4OD (International Conference on Information Technology for Organization Development), Tebessa, Algeria, 19–20 October 2014.

1 Introduction

Defined as a sub-field of artificial intelligence, the evolutionary computation has been used to develop controllers for autonomous robots, which developed a new computational discipline known as evolutionary robotics.

Researchers in biomechanics, robotics, and computer science try to understand human natural motion in order to find new inspirations to solve computation problems by reproducing this human motion. One of the most important fields which has been inspired by human motion is the humanoid robotic.

The main goal of humanoid robotic researches is to obtain robots that can imitate human behaviours in order to collaborate with humans in a best way. An obvious problem confronting humanoid robotics is the generation of stable and efficient gaits at a reasonable time. To deal with this problem, alternative bio-inspired control methods have been proposed like in Ouannes et al. (2012), which do not include any specification of reference trajectories helping to reduce computing time. More specifically, the best investment of the advanced computational systems is an opportunity to find new paths that provide efficient solutions to robotic issues.

Nevertheless, most of these robotic solutions are based on the classical artificial intelligence paradigm relying on human design, which are limited in terms of inefficiency if robot control is highly constrained and submitted to differentiable objective functions (Rubercht et al., 2011). Classical artificial intelligence is also unable to cope with uncertainty robot behaviours (Atyabi and Powers, 2013) due to the explicit programming of desired behaviours using a complete and an exact mathematical model to conceive the robot and its environment (Wang et al., 2012). In legged robots, various learning algorithms have been proposed to provide autonomous operations for many complex and challenging control and design problems such as Tang and ER (2007). One of such problems is bipedal walking robot control.

In order to address this issue, alternative biologically inspired control methods have been proposed (González-Nalda et al., 2011). To this end, training a multilayer neural network with an evolutionary algorithm (EA) is one of these efficient methods proposed to conceive robot controller. However, when controlling, the evolutionary-based neural network consumes often an important processing time, especially if there is a large amount of data, even with efficient neural network trainers. To cope with this restriction, parallel architectures have been proposed to enhance computing and processing performances (Petrovic, 2004; Sofge et al., 2007; Capi et al., 2005; Nageswaran et al., 2009; Luong et al., 2010), since the technologies of current processors (i.e., CPU-based architecture) have reached their limits as regards speed.

Modelling and integrating different modalities and behaviours requires complex algorithms and intensive calculations which could be held in a reasonable time.

The main challenge of simulation modelling is continuous evolution in terms of constraints and objectives. However, performing this continuous evolution takes many hours, even many days to be executed. This is heightened by the fact that these complex models can also have many additional parameters, which should be considered for its important role in the model behaviour. For these reasons, considerable efforts have been done into technologies, methodologies and theoretical advances in order to speed up execution without sacrificing model accuracy (Christely et al., 2010). Nevertheless, there were few studies to accelerate simulation of artificial intelligence performance using GPU architectures.

While central processing units (CPUs) were optimised to run sequential tasks, GPUs are devoted to calculate the floating point operations and to execute massively parallel tasks (Chu and Hsiao, 2014). Therefore, these recent advances have contributed to the occurrence of a new programming approach based on GPUs. Recently, many algorithms have been rewritten and redesigned for modern GPUs, which are characterised by a single instruction multiple data (SIMD) processing (i.e., massive parallelism). GPUs have been recently used to accelerate computations for various topics [e.g., neuroscience (Liu and Guo, 2013), evolutionary techniques (Jaros, 2012), medical image processing (Eklund et al., 2013), data compression (Patel et al., 2012), etc].

We note that the availability of new appropriate development environments such as CUDA and OpenCL tends to further simplify the development of parallel applications for this type of processors (Chu and Hsiao, 2014; Duran et al., 2014). At the same time, it allows the possibility of developing more integrative, detailed and predictive bio-inspired and artificial life models, while at the same time decreasing the computational cost to simulate those models.

In this study, our focus is to suggest an improved robot controller based on two bio-inspired techniques namely EAs and artificial neural networks (ANNs).

On one side, EAs are considered as one of the most important optimisation methods that generate approximate solutions, with extensive computational needs (Qin et al., 2012; Jaros, 2012). For their parallel nature, EAs are very suitable to distributed systems (Petrovic, 2004). Many of them aim at providing a general computational platform. As we are interested in this work by the graphic accelerators, a survey and a new classification of the related works about EAs on GPUs will be presented in the next section.

On the other side, several prior studies have examined alternative hardware ANN solutions using field-programmable gate array (FPGAs) and GPUs that deliver a fine-grained parallel architecture (Magure et al., 2007; Nageswaran et al., 2009).

The training of EA-ANN on the graphic platform accelerator is ideal, due to the features of NVIDIA's compute unified device architecture (CUDA) which is a programming framework for the massively parallel GPUs, also to the architecture of GPU that contains thousands of independent floating-point units connected to on-board memory enabling high memory bandwidth, making this device perfect for providing significant speed-up to the intensive parallel applications (Patulea et al., 2014), by respecting memory access rules.

As the general purpose computing graphic unit (GPGPU) is an emerging field in many domains. There were few approaches proposed to deal with applying this technique in evolutionary robotics (González-Nalda et al., 2011; Peniak et al., 2011; Shi et al., 2010; Montiel et al., 2014).

In this paper, we propose an original GPU-based approach in order to accelerate the training of an evolutionary robot. This is done to provide efficient behaviours required in several complex situations like walking toward a hidden destination. This proposal is based on the combination of an evolutionary strategy (ES) and recurrent neural network (RNN) where the ES is responsible for optimising the trajectory generation of humanoid robots, and the RNN forms the embedded brain of the robot. In order to speed up the evolution process of the RNN training, we propose the use of a GPU accelerator at multiple levels, taking into account the open dynamics engine (ODE) of GPGPU model proposed by Zamith et al. (2009). The process of the GPU-based robot controller executes order of magnitude times faster than the conventional CPU-based solution.

To achieve this, we address the effective use of the GPU memory hierarchy where the main novelty here is about reducing time transferring data between the two memory sub-systems (i.e., CPU memory and GPU memory). In addition, to improve GPU use, the judicious management of GPU parallelism is established for all components of our simulation or controller evolving. Furthermore, we provide a set of generic instructions for GPU programming that are advised to be followed in order to benefit of GPU performance. The reached results are compared with those obtained with the serial implementation of our algorithm by discussing the effects of a set of parameters related to the simulation or the evolution part.

The present paper is organised as follows. Section 2 introduces a set of basic concepts related to GPU computing using CUDA, followed by presenting the related works proposed to GPU-based robot controller modelling. A detailed description of our proposal (hybrid ES-RNN robot controller based on GPU) takes place in Section 3. The experimental study and discussion of the obtained results are presented in Section 4. Finally, Section 5 draws conclusions with some future research directions in this context.

2 CUDA model, ANNs, and EAs: background and related work

This section consists of a set of basic concepts related to the main tools and techniques of this work. The GPU programming paradigm as well as the physics simulator used to simulate the humanoid robot called ODE, have taken place in this section. Moreover, we will present in this section the GPU-based related works devoted to the robot controller modelling issues concerning the programming of such techniques are cited, namely threads distribution and data transfer issues. After that, we present some immersed physics simulators to graphic accelerators.

2.1 Basic concepts

2.1.1 CUDA model programming as a tool for parallel processing on NVIDIA cards

Nowadays, NVIDIA's CUDA technology provides a parallel computing architecture on modern NVIDIA's GPUs on which hundreds of streaming processors (SPs) are grouped into several streaming multiprocessors (SMs) with own flow control and on-chip shared memory units (Scalabrin et al., 2014; Duran et al., 2014). All SMs share a global memory with high latency. The number of SMs, the size of global memory, the number of SPs, the size of shared memory and the number of registers depend on GPU compute capability (Qin et al., 2012).

The sequential operations should be programmed as host functions that execute on CPU. Thus, parallel operations should be programmed as a kernel or device functions that execute on the GPU. Both host and kernel functions will be wrapped and called via a main host function.

In CUDA environment, the basic unit of a parallel code is a thread, thousands of threads can run concurrently with a same instruction set called a 'kernel'. A set of threads can be bundled into a grid of thread blocks with each block containing a fixed number of threads.

The communication between CPU and GPU can be done through global device memory, constant memory, or texture memory on a GPU board. After compilation by the CUDA environment, a program runs as a 'kernel' in a GPU. A kernel takes input parameters, conducts computations, and outputs the results to device memory where the result can be read by the CPU. With thousands of threads doing similar tasks simultaneously, the computation speed can be significantly improved. The CPU owns the host code that prepares input data and accepts output values from the GPU. The intensive computation task is handled by GPU kernels. The output data is written to global device memory in order to be retrieved by a CPU program. For fully comprehensive description, the reader is referred to the CUDA C Programming Guide (NVIDIA, 2011) and similar resources.

2.1.2 Open dynamics engine

ODE is an open source library for simulating rigid body dynamics and collisions in an efficient and accurate manner (Giovanni et al., 2011). It consists of a high performance library for the simulation of rigid body dynamics developed in a simple C/C++ API.

Although ODE is a good and reliable physics engine, computing all the physical interaction of a complex system can take a great deal of processing power. Since ODE uses a simple rendering engine based on OpenGL, it has restrictions for the rendering of complex environments comprising many objects and bodies. This can significantly affect the simulation speed of complex robotic simulation experiments (Bezák, 2012).

2.2 Related work

2.2.1 GPU-based ANNs for humanoid robot controller

ANN has been part of an attempt to emulate the learning phase of the human nervous system. However, the main difference arises from the fact that the nervous system is massively parallel (Prabhu, 2007), while the computer processor remains significantly sequential.

Since ANN requires a considerable number of vector and matrix operations to get results, it is very suitable to be implemented in a parallel programming model and run on GPUs.

CUDA has been employed in a wide variety of applications nevertheless, few of them investigate this technology for the evolutionary neuro-robotics domain. Only some of them have implemented to date CUDA to neural networks, as this field requires further investigation in applying the CUDA technology to neural computation evolved with evolutionary computation and robotics research (Peniak et al., 2011; Liu and Guo, 2013).

Peniak et al. (2011) presented a novel software application named 'Aquila', useful for cognitive and developmental robotics research. 'Aquila' addresses the need for high-performance robot control by using the latest parallel processing paradigm, based on the NVIDIA CUDA technology.

González-Nalda et al. (2011) aimed to create through evolution the neural network that couples with a complex humanoid robot body, where the neural network used was immersed in the CUDA programming model. In this work, the problems related to a non-structured environment and evolutionary robotics need a sub-symbolic conexionist approach based in nouvelle AI. This novel approach can cope with the coupling among sensorimotor, neural and environment parts.

Various types of neural networks are used to generate walking behaviours and control design of humanoid robots, where a few of them have investigated the GPU parallel processing. However, these ANN-based approaches did not allow building robot controller that fits with the complex humanoid robot body. To surmount this weakness, we propose in our current study the integration of the CUDA-based EAs to conceive the robot controller.

In the next subsection, we review the most important GPU-based EA research activities proposed in literature to enhance computing performance EAs.

2.2.2 GPU-based EAs

EAs are stochastic search algorithms allowing for the search area of alternative solutions to find the optimal one. It successes in addressing hard optimisation problems in diverse areas such as optimisation, learning, adaptation and others (Talbi, 2009).

The quality of the expected results depends on various factors, including the available computing power (Arora et al., 2010). The idea of EAs is derived from Darwin's evolution theory, where chromosomes are selected to form the parents. The creation of parents is followed by the selection step, in which chromosomes with better fitness can be selected with a higher probability. The selected chromosomes are reproduced using various operators to generate new offsprings. Finally, a replacement scheme is applied to determine which chromosomes of the population will remain from the offsprings and the parents, and so on until reaching a stopping criterion.

Algorithm 1 Evolution strategies template

```
Ensure: Best individual or population found.
1: Initialise a population of $\mu$ individuals;
2: Evaluate the $\mu$ individuals ;
3: repeat
4:     Generate $\lambda$ offsprings from $\mu$ parents;
5:     Evaluate the $\lambda$ offsprings;
6:     Replace the population with $\mu$ individuals
       from parents and offsprings;
7: until Stopping criteria
```

Source: Talbi (2009)

ESs represent one the most popular methods of the EAs. ESs are a particular kind of EAs such as genetic algorithms (GAs) or genetic programming (GP). In ES, solution (individual) representation is coded with vectors of real values. An individual (a chromosome) is composed of the float decision variables in addition to some other parameters guiding the search. For ES, an elitist replacement is used (Talbi, 2009).

Recently, due to their parallel nature, there have been several attempts to accelerate the EAs on the massively parallel GPU architecture (Luong et al., 2012).

This section reviewed various parallel EAs approaches using GPUs. We propose a classification of parallel EAs on GPUs based on the number of threads assigned for each chromosome (solution). Then, GPU-EA works can be distinguished into two categories: classification criterion is of some works of GPU-EAs is based on: chromosome-based category in which one thread is assigned to each chromosome, whereas each gene of the chromosome is attributed to each thread in the second one (gene-based category).

a Chromosome-based GPU-EAs

In this category, Arora et al. (2010) can be cited. This study presented an implementation based on CUDA toolkit for GAs. The authors studied the effect of a set of parameters (e.g., thread size, problem size or population size) on the performance of their GPU implementation compared to a sequential GA. The proposed method yield to 40 times faster than their serial counterparts.

Ogier et al. (2012) proposed porting different types of EAs on GPGPUs for easy specification of evolutionary algorithm (EASEA) framework which is dedicated to help non-specialists to optimise their problems by EA. The algorithms exposed, and the acceleration attainable in their work are indicated on different NVIDIA GPGPUs cards for different optimisation algorithm families.

Pospichal et al. (2010) adopted a parallel GA with an island model (Skolicki, 2005) for implementation running on GPU with 256 individuals in each island. To maintain population, the authors proposed to use an architecture based on mapping threads to individuals. The on-chip hardware scheduler is used in order to rapidly exchange existing islands between multiprocessors to hide memory latency. The authors report speeds-up up to 7,000 times higher on GPU compared to the CPU sequential version of the algorithm.
Jaros and Pospichal (2012) compared an optimised multi-core CPU implementation with a fine-tuned GPU version of a used GA for the knapsack Problem. The main objective of this study is to present the true performance relationship between modern CPU and GPU architecture and to eliminate some myths of GPU performance environment.

Zhu (2009) made his contribution toward the paralleling, and performance analysis of a massively parallel evolution strategy with pattern search (ES-PS) optimisation algorithm. His proposal exploits continuous optimisation functions dedicated to the optimisation of the acceleration provided by the graphics hardware. The obtained results indicate that GPU-accelerated ES-PS method is orders of magnitude faster than the corresponding CPU implementation.

b Gene-based GPU-EAs

Shah et al. (2010) is an example of this category. Authors of this work exploited the parallelism inside a chromosome, in addition to the parallelism performed to evaluate multiple chromosomes. Their study presented a generic framework for GAs speeded up by using modern graphic cards and tested on a variety of problems.

Krömer et al. (2011) proposed a technique using the same analogy with the previous one, but using the differential evolution (DE). With the help of the CUDA toolkit, the fitness was evaluated from 2.2 to 12.5 faster than on CPU using C code and from 25.2 to 216.5 times faster than on the CPU using object oriented C++ code.

Oiso et al. (2011) suggested a technique exploiting the parallelism between individuals and genes simultaneously. The proposed implementation method yielded approximately 18 times faster results than a CPU implementation of their benchmark tests.

Table 1 classifies the cited GPU-EA works into the proposed classification (i.e., chromosome-based and gene-based EA). These works are presented and compared in terms of data organisation, number of threads assigned to each chromosome,and the GPU memories used).

Table 1 Classification of EAs approaches: chromosome-based, gene-based GPU-EA

Classification: chromosome-based, gene-based GPU-EA				
	Study	*Data organisation*	*One/many threads-based approaches*	*Data memory management*
Chromosome-based	Arora et al. (2010)	Separate 1-D integer and float arrays.	Different threads of a block for different individuals of the population.	Variables of one type belong to different individuals are stored adjacently in arrays.
	Ogier et al. (2012)	Collection of objects representing individuals.	One thread per individual for the evaluation phase.	Individuals are grouped in a contiguous buffers.
	Pospical et al. (2010)	Separate 1-D arrays, where each array represent sub-population.	Every individual is controlled by a single CUDA thread.	The local island populations are stored in shared on-chip memory of GPU.
	Zhu (2009)	Separate 1-D arrays, where each array represent sub-population.	Every individual is controlled by a single CUDA thread.	Global, shared, and texture memory
Gene-based	Kromer et al. (2011)	The whole population can be seen as a real matrix.	Each vector (solution) is processed by a thread block.	The whole population resides in the main memory.
	Osio et al. (2011)	Collection of objects representing population.	Each individual is processed by an SM, and each gene to an SP.	The whole population resides in the main memory.
	Jaros et al. (2012)	C structure consisting of two one-dimensional arrays.	Each vector (solution) is processed by a thread block.	Chromosomes are stored in L1 cache of the GPU.
	Shah et al. (2010)	The whole population can be seen as a real matrix.	Each vector (solution) is processed by a thread block.	The population matrix resides in the main memory of the GPU.
	Benalia et al. (2015)	Collection of objects representing population.	Multiple CUDA threads work on one chromosome to evaluate its fitness.	Global, shared, and constant memory.

It is worth mentioning that all these studies exploited parallel aspect on GPU and showed the effectiveness of such devices to improve the evolution of complex behaviours. Therefore, to cope with the acceleration of the evolution of humanoid robot's complex behaviours, we propose to use parallel GPU-Oriented ES. The main objective of the ES is to deal with the training stage of the RNN which is responsible of the control for the humanoid robot's motion.

2.2.3 *Our evolutionary approaches against the cited approaches: a brief comparison*

Basically, most approaches of the literature are based on either the parallel evaluation of solutions on GPU or the execution of simultaneous independent/cooperative algorithms. These approaches can also be classified into fully parallelised approaches (i.e., all phases are parallelised) or partially parallelised approaches (i.e., just the evaluation phase is parallelised), where just a few of these works have exploited the parallelism between genes leading to two types of exploiting parallelism: inter-chromosome parallelism and intra-chromosome parallelism.

Concerning data organisation (i.e., the structure representation), arrays of 1D or 2D that collected the objects are generally used to store the population individuals. The majority of approaches proposed that an individual is composed of a genome in addition to other fields such as the fitness value and an 'already evaluated' Boolean variable. However, we have proposed in our evolutionary approach that the attributes of each robot (individual) are gathered in one data structure, including different physical simulation and evolutionary parameters (e.g., mass, positions of the ground contact in simulation world, dimensions of the used primitives, fitness value, statistical information (i.e., min/max fitness, average fitness) and, number of neurons and layers). Our implementation groups the individuals in contiguous buffers aiming to transfer all needed information in one single transfer of direct memory access (DMA) type. Also, the chromosome length is too large, where genes do not represent only values of the used ANN, but also their weights are considered. The ES parameters are stored in GPU constant memory, whereas the shared memory is used to facilitate the data access in the whole evolutionary process.

For the number of the assigned threads, most of the found implementations associate one thread to one individual (solution) for the master/slave model. Moreover, one $threads'$ block is assigned to one sub-population for the island model or the cooperative algorithms. However, the literature reader could find only few investigations that met the efficient managing of the threads parallelism with the memory constraints, especially when dealing with a large set of solutions or large problem instances. We note that these solutions are generally specific to a particular problem or to problems of the same family; we can mention here that some compilers are provided in order to optimise a naive code in CUDA framework. In our work, we propose to assign a block of threads for each individual with an efficient managing of the $threads'$ distribution on the card.

Concerning the data memory management, there are few explicit efforts that handle optimisation structures with the different available memories, which are strictly dependent on the studied optimisation problem. For these reasons, we have proposed our own memory management scheme taking into account all parts of the simulation process.

2.2.4 Threads distribution issue

Several studies have addressed the auto-tuning of sensitive parameters in CUDA, but, there were few works studying this point for EAs, among them we can mention two works (Ogier et al., 2012; Luong et al., 2012). The idea of the designed algorithm of Ogier is to hold the maximum possible SMs by maximising the threads assigned to each block taking into account both register and scheduling limitations. On the other side, Luong proposed a dynamic heuristic for GPU parameters auto-tuning of the local search method (LSM). As a result, the time measurement for each selected configuration, according to a certain number of trials, will deliver the best configuration parameters.

2.2.5 Data transfer issue

In a multiple device system, transfer of data between these devices is considered as the trickier thing to be coded, whether in the case of cluster programming or in coding for machines equipped of CPU or GPU. Transferring data takes many time and the programmer must be careful that the transfer time does not overpower any performance gains from parallelising an algorithm. In literature, this issue is covered by many studies (Baskaran et al., 2008; Pai et al., 2012) but a few of them have considered this point explicitly for EAs (Ogier et al., 2012; Luong et al., 2013).

To address the problem of data transfer time, we propose a sub-model as a memory access handler which orchestrates data transfers between CPU and GPU, that takes into consideration minimising CPU/GPU data transfers.

2.2.6 GPU-based physic simulators

Over the past decade, physics-based simulation has become a key enabling technology for different applications. It has taken a front seat role in computer games (Hirabayashi et al., 2012), animation of virtual worlds and robotic simulation. On the other hand, software packages for automatic controller design are not integrated (Todorov et al., 2012) with physics engines. As robotic hardware becomes more complex and efficient, the importance of simulation tools increases. Existing physics engines can be used to test controllers that are already designed.

Robots and physics engines are composed of a multitude of components requiring high accuracy in some areas, while in other domains often a precise CPU time is required. Hence, when studying this kind of machine, speed of simulation is the most important. To reach the requested acceleration, some physics engines are already using GPU accelerators (Joselli et al., 2008; Zamith et al., 2009) through CUDA or OpenCL in features like rigid body collision detection and particle interaction. Taking some of the physics calculations to the GPU is based on two main reasons: *first*, it is possible to take out some of the typical CPU computations, so it can perform other calculations and *second*, it is possible to optimise the physics engine in order to support more rigid bodies in the simulation.

Zamith et al. (2009) presented a modified version of the ODE simulator that takes the benefits of the graphics engines, focusing on the simulation of rigid bodies with some of its methods implemented on the GPU that yields a good result. A new method for the automatic process distribution between the CPU and the GPU for the presented physic engines has also taken place in their work. Despite, Zamith used the GPU as a math coprocessor in real-time applications in special games and physics simulations with the intention of supporting mathematics and physics, the proposed architecture is a valuable for the general purpose language or for Shader language and is tested on ODE.

In our study, we adopt GPGPU architecture proposed by Zamith et al. (2009) to ensure the use of the GPU as mathematics and physics coprocessor which accelerates mathematical and physical laws of the robot movement.

3 Hybrid ES-RNN robot controller based on GPU: design and implementation

This section explains the proposed hybrid ES-RNN robot controller based on the GPU. We start with an overview of our GPU-based proposal, then the model of the used robot (i.e., geometric primitives, joints, memory hierarchy used for robot primitives storage) is presented. The evolution of kernel parameters and data storage are introduced, followed by the illustration of the proposed architecture. Finally, we conclude this section by analysing the complexity of the proposed system as well as some GPU-based issues which have been considered in this proposal such as the task distribution and memory management.

3.1 Hybrid robot controller based on GPU: an overview

To cope with the critical sides of the GPU programming paradigm, with the aim of getting a considerable speed's gain of the evolution of our controller, we propose a new hybrid architecture based on ES and RNN which is conceived on a GPU platform. This model adopts the GPGPU architecture proposed by Zamith et al. (2009) in order to exploit the high accuracy of this kind of device. In that case, the GPU is used as a math co-processor with the intention of supporting mathematics and physics, in the physic simulation side.

Concerning RNN specification, the studied humanoid robot uses Elman RNN for its biological plausibility and powerful memory capabilities. Despite biological neural networks do not make use of back propagation of learning, we propose to use GPU-based ES to evolve locomotive behaviours. For this task, we must exploit a big RNN to connect and to simulate muscles as a proprioceptive and a motor system in a humanoid robot with tens of degrees of freedom. The neural network must possess more than thousands of neurons because it is easier to extract and to set the correct information for each joint. The number of nodes in the hidden layers of the RNN is limited by the GPU memory capacity. This hybridation (RNN + ES) exploits the parallelism at a higher level; groups of threads are formed to handle a single chromosome which corresponds to each robot.

The purpose of the ES is to optimise the weights of the RNN which controls the humanoid robot walk. A synergistic relationship exists between the ES and the RNN as shown in Figure 2. The ES optimises the RNN, and the RNN produces robot behaviour that is then ranked. At start-up, the simulation is launched on the CPU side, the ES parameters are copied to the GPU constant memory, positions of the feet of robots are copied to the GPU global memory, the population chromosomes are initialised to random with one (01) gene per RNN weight. The number of connections represents the number of genes in the chromosome; a floating-point number represents each gene. Features values for all evolution training instances are transferred into the GPU main memory to be prepared for ES-RNN computations. At each simulation step, the device will execute the phases described in Figure 4.

After the initialisation on GPU side and for each generation, all robot sensors are activated in order to gather information from the simulation environment. These values represent the RNN inputs. As a result, the RNN outputs will be updated using our parallel evolutionary proposal. To achieve this, the distribution of the element tasks (threads) is run in a judicious manner using the proposed distribution algorithm and the efficient management of the available hierarchy memory. Furthermore, the new population (i.e., gathered from the last generation) represents the inputs of the new RNN. We can notice here that the sensors in Figure 2 represent the parts in direct contact with the simulation environment. These sensors deliver data coming from the environment like the ground contact, whereas the effectors have the ability to interpret the $outputs'$ RNN values (i.e., muscles which are represented here by some geometric primitives). The Kernels mentioned in GPU side represent the evolutionary computation running on the device, from the initialisation to the statistical kernel.

Technical details concerning implementation of the GPU-ES, start with the ES initialisation of the first population which is done in parallel manner. It means that parallel initialisation of all the RNNs is efficient because the occupation rate of the graphics cards is a factor that directly affects the overall performance.

The GPU evolutionary process simulation is described in the pseudo-code 2. Thereafter, next subsections will describe the mapping of the population on the card and details of $kernels$' implementation.

3.2 *Sample of demonstration: humanoid robot model and data storage*

Our demonstration is based on a model of robot that was created from primitives available in the ODE simulation package, which deliver a controlled environment with or obstacle.

Algorithm 2 The simulation process on GPU

```
 1: For Each Step of Simulation
 2: Upload the simulation parameters and
    initialisation of the robots;
 3: RNN initialisations;
 4: Generate the initial evolutionary process
    solutions (population);
 5: ES initialisations;
 6: Allocate problem inputs on GPU memory;
 7: Allocate the population on GPU memory;
 8: Allocate the fitness structures on GPU memory;
 9: Copy problem inputs on GPU memory;
10: Copy the generated population on GPU memory;
11: Copy additional structures on GPU memory;
12: Evaluate the fitness values of all individuals of the
    population by using a fitness measure based
    on the objective function to be optimised;
13: repeat
14:    for all all individuals in parallel do
15:       Evaluation of the individual;
16:       Insertion the resulting fitness into the
          fitness structures;
17:    end for
18:    Copy the fitness structures on CPU memory;
19:    Apply the recombination operators;
20:    Construct the new population (Robots);
21:    Replace the old population by the new one;
22:    Copy the new population on the GPU memory;
23:    Copy additional structure on the GPU memory;
24: until a stopping criterion satisfied
```

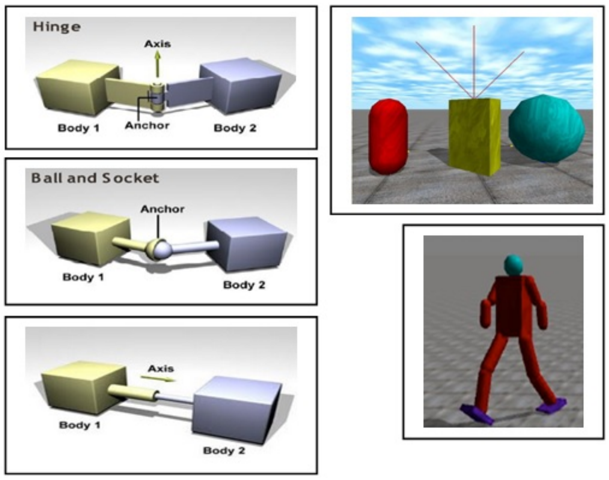
A physically-based model of bipedal locomotion describes the nonlinear relationships between the forces, the moments acting at each joint, the feet, the position, the velocity and the acceleration of each joint angle. In addition to the geometrical data, a dynamic model requires kinematical data mass, center of mass and inertia matrix for each link and joint, max/min motor torques and joint velocities, which are difficult to be obtained and are often an overlooked source of simulation inaccuracy. All these data are copied into the main memory of the GPU in order to make the largest possible computation on it.

To simulate interaction with the environment, the detection and handling of collisions as well as the suitable models of foot ground contacts are required. To deal with the simulation world and after performing several trials, we set the experimental parameters in Table 2, as the best empirical values of our legged robot.

Table 2 Body parameters of the robot

Body part	*Geometry primitive*	*Dimendion (m)*
Head	Sphere	Radius:0.188
Arm	Caped cylinder	0.14 × 0.25 × 0.44
Torso	Rectangular box	0.9 × 0.25 × 1.0
Thigh	Caped cylinder	0.20 × 0.25 × 0.7
Shank	Caped cylinder	0.20 × 0.25 × 0.7
Foot	Rectangular box	0.4 × 0.5 × 0.1

Figure 1 Geometrics primitives, joints, and the 3D humanoid robot (see online version for colours)



3.3 *Kernels invocations, ES population and data storage parameters*

With so many different types of memory, incorrectly using of the available memory in the device can affect the desired speed-up. So, generally the asked question concerns the correct type memory that must be used in each phase of the implementation. Once the structure created in the host side, it will be copied to the GPU constant memory, as this type is used for cases of data that will not change over the execution of the kernels ES. This structure includes population size, chromosome size, mutation rate, crossover rate, number of generations, statistical data (e.g., average fitness, total fitness, best individual), etc.

Our ES population is laid out in main memory of the GPU, as a two dimensional N × L matrix, where columns refer to chromosomes and rows correspond to genes within chromosomes. Here, N is the population size and L is the chromosome length, taking into consideration that storing variables of one individual sequentially in an array does not permit efficient memory access. So, to reach the coalescing memory, variables of one type and owned by different individuals of the population are stored adjacently in buffers.

3.4 *Architecture of the hybrid GPU-based ES-RNN robot controller*

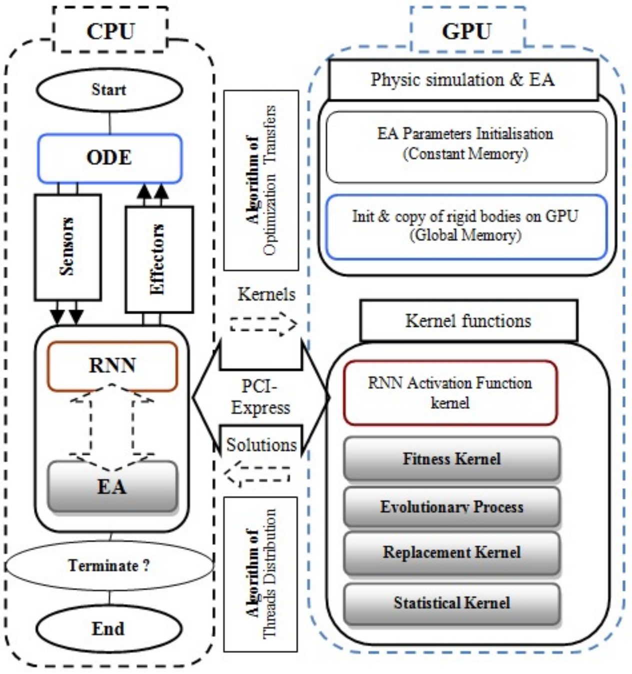
As mentioned above, the GPU in such applications is used as a co-processor, where the high computational phases are assigned to the GPU such as the RNN evolutionary process with some physical simulation calculations.

To conceive an efficient implementation of the genetic manipulation kernel, a low divergence of memory accessing and enough data to use all CUDA cores are required.

3.4.1 *Parameter configuration of the kernels*

To find the best parameters configuration (block shape, total number of threads) of our evolutionary kernels, we propose and apply an algorithm which measures the time of each configuration. The final objective of a such algorithm is to choose at the end the most appropriate one based on the time taken for each configuration that maximises, at the same time, the occupancy of the used card.

Figure 2 Configuration of RNN and ES with ODE on GPUs (see online version for colours)


CPU
Start
ODE
Sensors
Effectors
RNN
EA
Terminate ?
End
Algorithm of Optimization Transfers
Kernels
PCI-Express
Solutions
Algorithm of Threads Distribution
GPU
Physic simulation & EA
EA Parameters Initialisation (Constant Memory)
Init & copy of rigid bodies on GPU (Global Memory)
Kernel functions
RNN Activation Function kernel
Fitness Kernel
Evolutionary Process
Replacement Kernel
Statistical Kernel

The principle of our distribution strategy proposes that the number of blocks (number of chromosomes) should not be greater than the MPs. Moreover, the number of threads is the number of weights in each RNN, where the block capacity of the used card should be respected (the max block size).

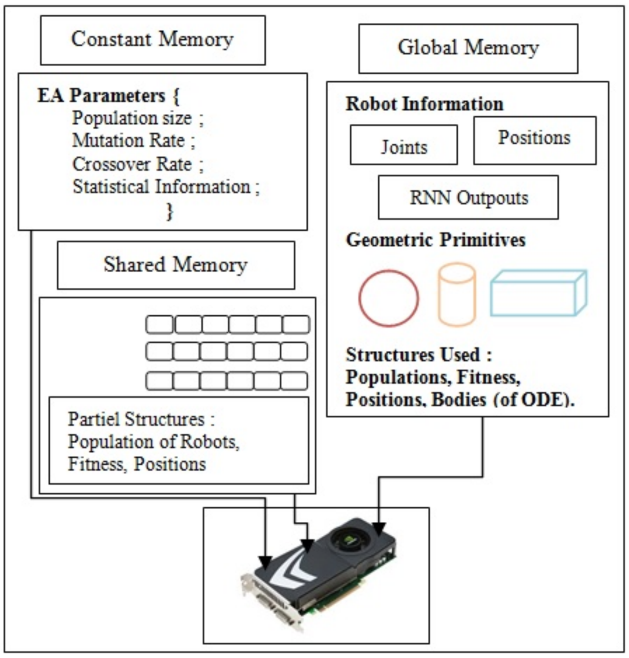
3.4.2 *Memory management*

It is mandatory to minimise CPU-GPU data transfers, since there is an important information exchanged between the proposed modules (which are simulation, nervous system, evolution training), these modules will be explained in detail in the next subsection. We propose to minimise CPU-GPU data transfer by sub-module inspired from Becchi et al. (2010) and Yang et al. (2012), taking advantages of the rules of optimising global memory accessing through coalescing, and scheduling capability to overlap memory latencies. GPGPU programmers consider that the coalescing (Yang et al., 2012) refers to the need that the accesses from a half warp (i.e., 16 consecutive threads can be coalesced into a single memory access).

To minimise the rate of transfers between the two subsystems (CPU and GPU), our proposal is based on the memory coalescing. The basic idea of the evolution process CPU-GPU memory mapping is the vectorisation of the used data structures (i.e., population structure, fitness structure, etc.) helping to save all data on vectors or 2D matrix to facilitate the coalescing process. Therefore, the mapping of data on GPU memory is performed using CudaMalloc, the transfer of data is ensured using CudaMemcpy where we check before any transfer, if this data resides on the GPU memory or not. Before executing training evolutionary kernels, the data coalescing is checked using rules defined by Yang et al. (2012).

At the end of this process data is de-allocates data using CudaFree, either at the end of the application or when the memory of GPU is full.

Figure 3 Memory management of the simulation process (physic motor, evolutionary training) (see online version for colours)

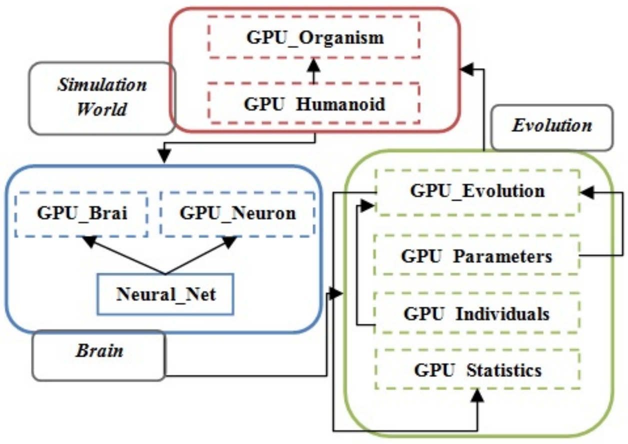


3.4.3 *Hybrid GPU-based ES-RNN modules*

Our system is equipped with three modules, where the first one is the simulation world module which is responsible of simulation task in ODE environment taking into consideration all required physics laws. The second one is the brain module, which creates the robot brain using the RNN. The last one is the principle module (evolution module) of our system in which all ES phases are implemented on GPU as kernels.

In the next subsections, kernels of the evolution module will be detailed.

Figure 4 The main classes of the proposed model (see online version for colours)



3.4.3.1 *Random number generation*

One of the factors which affect the performance of EAs is the generation of random numbers because theses algorithms are stochastic search processes.

However, CUDA libraries do not include any functions of a random number generator (RNG) at present, despite the fact that RNG is naturally necessary for executing GA processes. In order to generate random numbers in our application, we use the Random123 library of ‘counter-based’ random number generators (CBRNGs) as described in Salmon et al. (2011). The Random123 library can be used with the CPU (C and C++) and the GPU (CUDA and OpenCL) applications. This library is chosen due to the fastness of its generator, easier to parallelise, use minimal memory/ cache resources (three times faster than the standard C rand function and more than ten times faster than the CUDA cuRand generator).

3.4.3.2 *Population initialisation kernel*

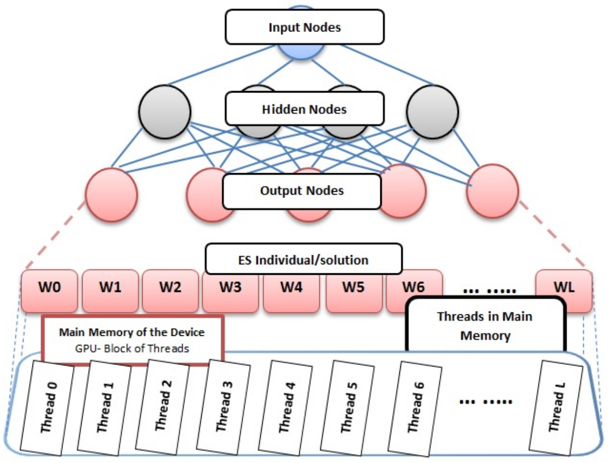
In each evolutionary process, the first step is the initialisation of the population. One of the questions facing this step, is where the chromosomes will be generated. For responding this exigency two mainly approaches are proposed in the literature (Luong et al., 2013):

- Generation of the population (chromosomes) on the CPU and its evaluation on the GPU: at each iteration of the evolution process, the population is generated on the CPU side. Its associated structure storing the solutions is copied on GPU. There by the data transfers are essentially the set of population solutions copied from the CPU to the GPU.
- Generation of the population (chromosomes) and its evaluation on the GPU: in this approach the population is generated on GPU. It implies that no explicit structure needs to be allocated. Thereby, only the representation of the solution must be copied from the CPU to the GPU. The benefit of such an approach

is to reduce drastically the data transfers, since the whole population does not have to be copied.

In our case and as mentioned above, the elements of each chromosome represent the weights of the RNN, the operation of getting these values is fully parallelisable and all chromosomes are initialised at the same time. This phase is performed by block of threads for each chromosome, as shown in Figure 5.

Figure 5 Block threads chromosome mapping (see online version for colours)



3.4.3.3 Selection kernel

Since most selection schemes work with probabilities and conduct random sampling of the population, and even promote multiple sampling of the same individual, the samplings can be done in parallel (Hofmann et al., 2013).

We note that the modified form of the roulette wheel selection is used in our selection phase. There are few works that contribute to the implementation of roulette wheel selection on the GPU. For example, Dawson and Stewart (2013) proved that a highly parallel roulette selection algorithm must respect the exploiting of warp-level parallelism, reducing shared memory dependencies.

To simulate the roulette wheel selection kernel in our case, the $robots'$ population is sorted (GPU method). This fitness is calculated based on the covered distance, using the fast GPU-based radix-sort, provided and described in Satish et al. (2009). These score values are standardised to calculate selection probabilities. We contribute in this phase by using thread block for each chromosome in the process. A reduction sum is performed on the standardised array which contains the gathered values. This new array is stored in global memory and used as a roulette wheel array.

3.4.3.4 Evolutionary process kernel

This kernel aims at updating the old population, by selecting all pairs of chromosomes undergoes the process of crossover to produce offsprings. We contribute in this phase by applying a massively parallel one point crossover, and mutation; these two operators are controlled by crossover and mutation probabilities.

In order to implement one point crossover, we need to two kernels, the first one is about finding the crossover point. We suggest here to assign one thread to one chromosome in order to solve the synchronisation problem between CUDA blocks. This phenomenon occurred in the case of assigning a thread block to each chromosome, this means that all threads must have the information about the crossover point. The second one is about getting back the new offsprings. To do that, Each CUDA block is organised as two dimensional. The x dimension corresponds to the genes of a single chromosome, while the y dimension corresponds to different chromosomes. The entire grid is organised in two dimensions with the x size fixed at 1, and the y size corresponding to the offspring population size divided by y block size multiplied by 2.

For the mutation kernel, the distribution of threads is used, where each thread occupied gene, this process has an objective to remote the ES out of local maxima or minima.

3.4.3.5 Fitness function kernel

In this proposal, a fully parallel method is proposed for the evaluation process in order to reduce computation complexity, since this step is the most important, which consumes more time as any EA. In our case, the best individual is the robot that is able to travel the largest distances in a given time. In this problem, the fitness function is based on the initial and the final positions of each robot (individual). The same decomposition as the evolutionary process is used, where each warp is copied to shared memory, as this memory can provide a great speed-up by conserving bandwidth to global memory. This advantage is due to its user-managed cache characteristic (NVIDIA, 2011) when calculating the distance in the GPU.

3.4.3.6 Replacement kernel

This phase uses Roulette Wheel selection over the parents and offspring to create the new parent population. The used kernel parameters are the same as they used in the previous phases with one modification which is that the dimensions of the kernel are derived from the parent population size.

3.4.3.7 Statistics kernel

In each generation, after the calculation of the fitness values, the statistics need to be updated. The statistics of the population is used for the selection process and for the termination of the decision. The maximum and minimum values of the fitness function, the average and the deviation constitute the structure of the statistical data.

3.5 Complexity analysis

Three parts are considered to calculate the computational complexity of this proposal: the simulation part, the nervous system part and the evolutionary part.

In the simulation part, the ODE computational complexity is of O(n^2), where n is the number of objects (geometric primitives). Moreover, the neural network complexity is of O(2^{l-1}) where l is the number of neural network hidden layers. Regarding the evolution part, the complexity depends on the evolutionary steps and operators. As the used operators are roulette wheel selection, one-point mutation and one-point crossover, the evolutionary part complexity is of O $(g \cdot (t \cdot m + t \cdot m + n))$ = O $(g \cdot t \cdot m)$; with g is the generation number, t is the population size and m is the individual size.

As a result, the total computational complexity of the proposed algorithm is calculated by summing up the three parts complexity: O(n^2) + O (n^2) + $O(g \cdot t \cdot m)$.

As the selected RNN is of three hidden layers, it is complexity is negligible against the others. Consequently, our proposed system is of a quadratic complexity equal to O $(g \cdot t \cdot m)$.

4 Experiments, results and discussion

We remind that the first goal of this study is to demonstrate the potential of using GPU devices for the ES used to evolve a RNN. To achieve this, we have performed various experiments by changing various ES parameters, namely the chromosome length, population size, number of generations, and the sensitive parameters of the executed kernels (block shape, number of threads). The gain has been measured over the entire ES. In our tests, ES phases are timed on the CPU and the GPU. For more accuracy, the population transfer time, to and from the GPU, is added to the kernel's execution time.

4.1 Experimental setup and execution environment configuration

We have implemented the proposed model using C/C++ programming language and CUDA (4.0) framework. These experiments are run on a PC with one processor (Intel Core i7 870) and one GPU (NVIDIA GeForce GTX480). The used humanoid model is a fully three-dimensional bipedal robot with 15 degrees of freedom, 12 rigid-body parts, and 11 ODE joints. For the ES, we have used a crossover rate fixed at 0.7, genomic mutation rate equal to 0.01, elitism rate of 0.2, roulette wheel selection method and at least 100 individuals for each population. The simulation is performed until having the adequate movements of the robots.

4.2 Convergence of the evolutionary proposal

First and foremost, any parallelisation effort must ensure that final reached solution is similar or better in quality to that obtained from the serial algorithm. First, we will demonstrate that our parallel evolutionary approach is able to find a solution which is very close to the optimal one. Specifically, the expected optimal solution is related to the best RNN weight values that conduct robot to move like a real human to perform complex tasks.

The convergence of the algorithm can be studied by investigating the average objective function values of the whole population generation. To discuss the algorithm convergence, we consider the average values of the objective function, and its taking average objective values, minimum and maximum values for 1,000 different runs of the ES as shown in Figures 6 and 7. We can remark that our proposal has reached the same convergence compared to the serial approach, with a value of 93.76 as a fine forward walk with a slight limping gait.

Figure 7 assures our confidence that our GPU-based ES performs better than its similar CPU-based at least 95% of the time.

Figure 6 Fitness values over different generations on CPU and GPU (see online version for colours)

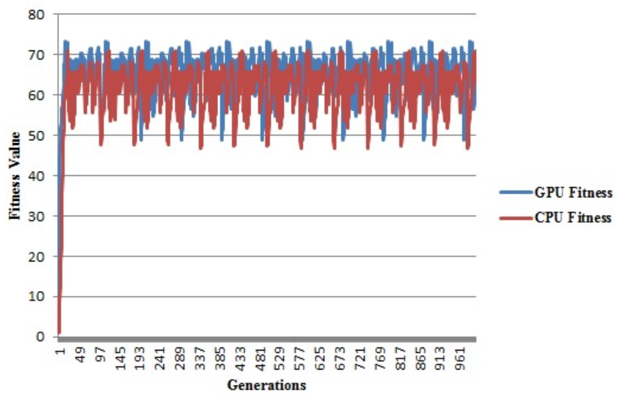
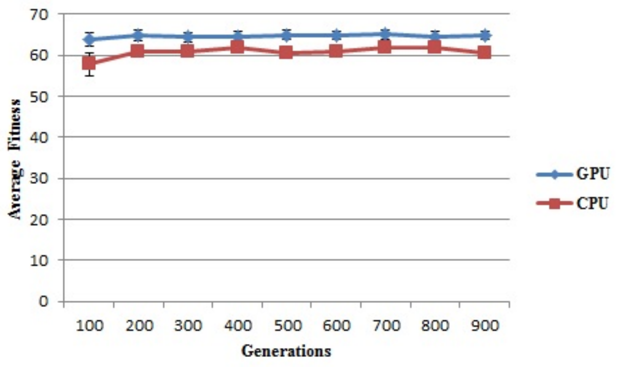


Figure 7 The average fitness with a 95% confidence interval for each algorithm (see online version for colours)



4.3 Impact of the ES parameters and the CUDA parameters on evolution speed-up

Having demonstrated similar convergence characteristics, we are now ready to study the scale-up property of the algorithm by measuring the effect of the number of decision variables, the population size and the problem complexity. Each run is finished after each generation executed over a fixed period defined by the ODE. The gain value is computed by dividing the overall computational time of serial implementation with respect to the time taken by the parallel implementations. The obtained results showed that when the number of decision variables increases, the speed-up of the parallel ES increases. Also, the speed-up increases according to the population size growth. Interestingly, in point of view complexity, the speed-up is smaller for more complex problem, but as the population size is increased the speed-up gets more or less identical. For example, if we have ten individuals in the population (which means 12 * 10 = 120 geometric primitives), the order of acceleration is about 1.11X, with 50 (600 geometric primitives). In the case of 100 individuals (1,200 geometric primitives), the gain is improved seven times to 12X and more.

Figure 8 shows the experimental result of the GPU-based and CPU-based implementations on different generations of the EA. The global performance of our system increases according to the augmentation of individual number of the population as well as the number of generations as shown in Figures 9 and 10.

As a conclusion, our hybrid ES-RNN robot controller based on the CUDA toolkit has demonstrated that it can be 12 times faster than a common CPU-based robot controller when simulating the evolving Elman neural network model.

Figure 8 Global system execution speed-up

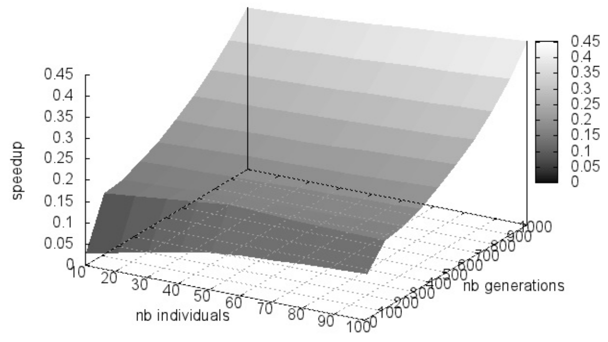


Figure 9 CPU vs. GPU timing of EA phases (see online version for colours)

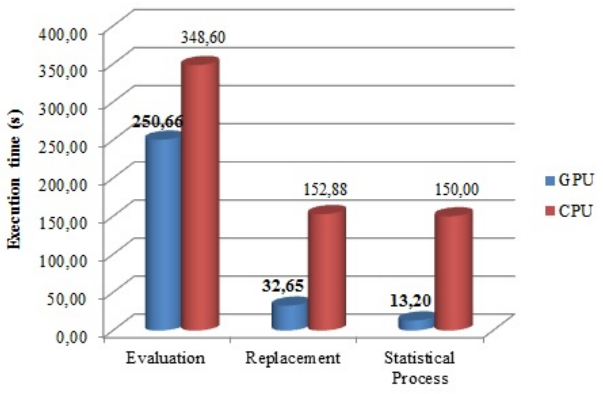
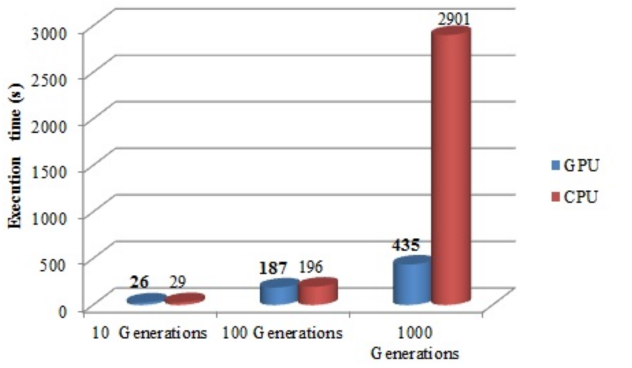


Figure 10 System execution time on CPU and GPU over different generation (see online version for colours)



4.4 *Performance analysis and solution quality, under time limits*

Figure 9 depicts the experimental results of the GPU and CPU-based implementations on different phases of the proposed ES. These outcomes have been reached after 100 tests. We note that parallelising the process of both chromosomes and their genes is more effective, as each gene is mapped to one thread in the block which holds the chromosome. This is due to the fact that the GPU-based implementation enables the execution of more threads. In addition, the execution of the most ES processes can minimise the frequency of data transfer between the host and the device, which is the biggest challenge in such application on the GPU.

After discussing the differences between the results obtained by GPU-based and CPU-based implementations, we discuss in this subsection the impact of some parameters on the entire performance. The devoted time and the memory occupation (shared memory or registers) to the distance function is longer than the other Kernels cases of in the two chromosome sizes (n = 512) and (n = 1,024). The reason behind this phenomenon is that the elements of calculating the fitness kernel are coming from the physic simulation performed by ODE. The global performance of our system increases according to the augmentation of individual number of the population.

4.5 *Performance discussion in terms of efficiency and effectiveness*

For each step of the simulation, a mono-core CPU-based implementation, CPU-GPU, GPU-CPU exchange data among memories are considered. The average time has been fixed at 100 trials. The average values of evaluation function have been collected and the number of successful tries is also represented.

Since the computational time can reach 1,000 generations and more, the average expected time for the CPU-based implementation has been deduced on the basis of two execution trials.

The generation and the evaluation of the chromosomes in a parallel manner on GPU architectures provide an efficient way to speed up the evolution and the search process in comparison with a CPU-based version. As shown in Figure 10, GPU-based version has been already faster than the CPU one (i.e., GPU-based order of acceleration is 12 times faster than CPU-based implementation). Due to high misaligned accesses to global memories, non-coalescing memory reduces the performance of the GPU-based implementation. To overcome the problem and reach the coalescing memory, variables of one type belong to different individuals of the population are stored adjacently in buffers. GPU keeps accelerating the hybrid evolutionary process as long as the size increases. Concerning the solution quality, the obtained results of the proposed hybrid ES are quite competitive compared to those obtained with the sequential ES.

Figure 11 shows how fast every generation is evaluated, whereas Figures 12 and 13 represent respectively the medium time for a couple of generations compared to the population size and the time evolution CPU vs GPU over generations depending on the number of individuals, through an execution on the CPU and the GPU in order to show more information about the execution, as mentioned above the objective function in this case is the distance covered by each robot where all the individuals starts and finishes at the same time (in each simulation step). In each generation, a good phenotype is a set of parameters that causes a character to perform the desired movement.

Figure 11 Time evolution over generations (see online version for colours)

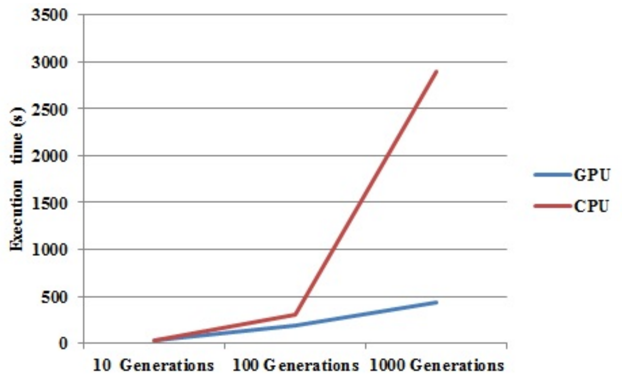
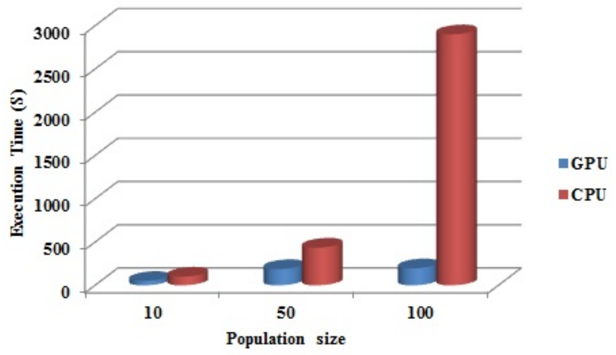
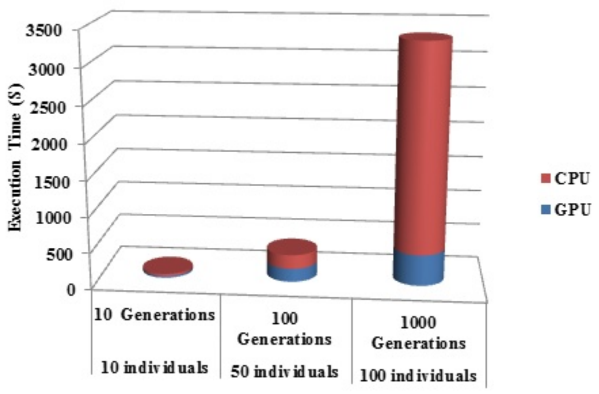


Figure 12 The medium time for every generation compared to the population size (see online version for colours)



We conclude that the use of GPU provides an efficient way to deal with this kind of application where a lot of parameters are considered (e.g., simulator parameters, RNN parameters, and EA parameters). Consequently, implementing training on a GPU has allowed exploiting parallelism in such application and improving the robustness/quality of the provided solutions. On the other hand, as this problem takes into account a bench of parameters, the order of acceleration achieved is very promising to go throw the optimisation methods, tuning the ES parameters on the GPU device, using the multi-GPU and the new architectures as Kepler, or others.

Figure 13 Time evolution CPU vs. GPU (see online version for colours)



5 Conclusions and future work

The purpose of this article is to explore the feasibility of the use of GPU technologies capabilities to simulate a complex model using artificial bio-inspired approaches based on ES and RNN. To achieve this, a novel hybrid (ES-RNN) approach was proposed to construct and simulate a 3D model of locomotion for a humanoid robot. More specifically, movement and evolution brain of a humanoid robot is conceived using the proposed model, which is simulated on a physics simulator aiming to imitate their physical phenomena such as the gravity, the collision, etc. We have implemented our evolutionary model of the RNN in GPU-based platform and we have discussed various programming issues. Moreover, we have provided a set of design guidelines that could be useful for other modellers, in the way to avoid common pitfalls such as the available memory limitation on GPU architectures, and the synchronisation between threads among blocks, as well as to exploit GPU architectures for performance gain.

In fact, it is more likely to assume that the RNN evolution training and simulation actions operate on a much longer time scale than other operations, which allow a time scale separation. This work has successfully performed the cited steps by implementing an optimised brain (controller) that can distribute some of its processing between GPU and CPU, allowing the developer of an automatic model to decide where to process its computing steps.

As future work, we propose the improvement of the accuracy of our technique by incorporating tuning parameters of the kernels and testing other ANNs with other evolutionary training methods. Additionally, parallelisation techniques of EAs involving multiple populations may interact favourably with such type of applications. Separate populations (GPU-based EA island model) may be trained on different subsets of the training data, allowing more optimal searches of the research space.

References

Arenas, M.G., Mora, A.M., Romero, G. and Castillo, P.A. (2011) 'GPU computation in bioinspired algorithms: a review', *Proceedings of the 11th International Conference on Artificial Neural Networks Conference on Advances in Computational Intelligence*, Springer-Verlag, pp.433–440.

Arora, A., Tulshyan, R. and Deb, K. (2010) 'Parallelization of binary and real-coded genetic algorithms on GPU using CUDA', *IEEE Congress on Evolutionary Computation (CEC '10)*, pp.1–8.

Atyabi, A. and Powers, D.M. (2013) 'Review of classical and heuristic-based navigation and path planning approaches', *International Journal of Advancements in Computing Technology*, Vol. 5, No. 14, pp.1–14.

Azamat, M., Daniel, L., Ching-Chen, M. and Boyana, N. (2012) 'Autotuning stencil-based computations on GPUs', *IEEE International Conference on Cluster Computing (CLUSTER '12)*, IEEE, pp.266–274.

Baskaran, M.M., Bondhugula. U., Krishnamoorthy, S., Ramanujam, J., Rountev, A. and Sadayappan, P. (2008) 'Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories', *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, New York, pp.1–10.

Becchi, M., Byna, S., Cadambi, S. and Chakradhar, S. (2010) 'Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory', *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ACM, New York, pp.82–91.

Bezák, P. (2012) 'GPU accelerated robotic arm optimal trajectory generation using genetic algorithms', *Transfer inovácií [elektronický časopis] – Košice: Inovačné centrum automobilovej výroby*, pp.44–45.

Capi, G. and Doya, K. (2005) 'Evolution of recurrent neural controllers using an extended parallel genetic algorithm', *Robotics and Autonomous Systems Journal*, Elsevier, Vol. 52, No. 2, pp.148–159.

Christely, S., Lee, B., Dai, X. and Nie, Q. (2010) 'Integrative multicellular biological modeling: a case study of 3D epidermal development using GPU algorithms', *BMC systems biology Journal, BioMed Central Ltd.*, Vol. 4, No. 1, pp.107.

Chu, S.L. and Hsiao, C.C. (2014) 'Optimising space exploration of OpenCL for GPGPUs', *International Journal of Computational Science and Engineering*, Inderscience, Vol. 9, No. 1, pp.64–79.

Dawson, L. and Stewart, I. (2013) 'Improving ant colony optimization performance on the GPU using CUDA', *IEEE Congress on Evolutionary Computation (CEC '13)*, IEEE, pp.1901–1908.

Dinkelbach, H. Ü., Vitay, J., Beuth, F. and Hamker, F.H. (2012) 'Comparison of GPU-and CPU-implementations of mean-firing rate neural networks on parallel hardware', *Network: Computation in Neural Systems Journal*, Informa UK, Ltd. London, Vol. 23, No. 4, pp.212–236.

Duran, R.E., Chen, D., Saraswat, R. and Hallmark, A. (2014) 'A framework for comparing high performance computing technologies', *International Journal of Computational Science and Engineering*, Inderscience, Vol. 9, No. 1, pp.119–129.

Eklund, A., Dufort, P., Forsberg, D. and LaConte, S.M. (2013) 'Medical image processing on the GPU – past, present and future', *Medical Image Analysis Journal*, Elsevier, Vol. 17, No. 8, pp.1073–1094.

Giovanni, S. and Yin, K. (2011) 'LocoTest: deploying and evaluating physics-based locomotion on multiple simulation platforms', *Proceedings of the 4th International Conference on Motion in Games (MIG '11)*, Springer, pp.227–241.

González-Nalda, P. and Cases, B. (2011) 'Topos 2: spiking neural networks for bipedal walking in humanoid robots', in Emilio Corchado, Marek Kurzyński and Michał Woźniak (Eds.): *Proceedings of the 6th International Conference on Hybrid Artificial Intelligent Systems – (HAIS '11)*, pp.479–485, Springer-Verlag, Berlin, Heidelberg.

Hirabayashi, M., Kato, S., M-Edahiro, M. and Sugiyama, Y. (2012) 'Toward GPU-accelerated traffic simulation and its real-time challenge', *REACTION, Proceedings of First International Workshop on Real-time and Distributed Computing in Emerging Applications*, San Juan, Universidad Carlos III de Madrid.

Hofmann, J., Limmer, S. and Fey, D. (2013) 'Performance investigations of genetic algorithms on graphics cards', *Swarm and Evolutionary Computation Journal*, Elsevier, Vol. 12, pp.33–47.

Jaros, J. and Pospichal, P. (2012) 'A fair comparison of modern CPUs and GPUs running the genetic algorithm under the knapsack benchmark', *Applications of Evolutionary Computation Book*, Springer, pp.426–435.

Jaros, J. (2012) 'Multi-GPU island-based genetic algorithm for solving the knapsack problem', *IEEE Congress on Evolutionary Computation (CEC '12)*, IEEE, pp.1–8.

Joselli, M., Clua, E., Montenegro, A., Conci, A. and Pagliosa, P. (2008) 'A new physics engine with automatic process distribution between CPU-GPU', *Proceedings of the ACM SIGGRAPH Symposium on Video games (Sandbox '08)*, ACM, New York, pp.149–156.

Krömer, P., Snåšel, V., Platoš, J. and Abraham, A. (2011) 'Many-threaded implementation of differential evolution for the CUDA platform', in Natalio Krasnogor (Ed.):*Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation (GECCO '11)*, pp.1595–1602, ACM, New York, USA.

Liu, J. and Guo, L. (2013) 'Implementation of neural network backpropagation in CUDA', *Intelligence Computation and Evolutionary Computation, Advances in Intelligent Systems and Computing*, Vol. 180, pp.1021–1027.

Luong, T. V., Melab, N. and Talbi, E. (2010) 'Parallel hybrid evolutionary algorithms on GPU', *IEEE Congress on Evolutionary Computation (CEC '10)*, pp.2734–2741.

Luong, T.V., Melab, N. and Talbi, E. (2013) 'GPU computing for parallel local search metaheuristic algorithms', *IEEE Transactions on Computers Journal*, IEEE, Vol. 62, No. 1, pp.173–185.

Magure, L.P., McGinnity, T.M., Glackin, B., Ghani, A., Belatreche, A. and Harkin, J. (2007) 'Challenges for large-scale implementations of spiking neural networks on FPGAs', *Neurocomputing Journal*, Elsevier, Vol. 71, No. 1, pp.13–29.

Montiel, O., Sepúlveda, R. and Orozco-Rosas, U. (2014) 'Optimal path planning generation for mobile robots using parallel evolutionary artificial potential field', *Journal of Intelligent & Robotic Systems*, Springer, Vol. 52, No. 2, pp.1–21.

Nageswaran, J.M., Dutt, N., Krichmar, J.L., Nicolau, A. and Veidenbaum, A. (2009) 'Efficient simulation of large-scale spiking neural networks using CUDA graphics processors', *International Joint Conference on Neural Networks (IJCNN '09)*, IEEE, pp.2145–2152.

CUDA Toolkit 4.0 CURAND Guide (2011) NVIDIA Corporation, Version 12.3, January.

Ogier, M., Krüger, K., Querry, S., Lachiche, N. and Collet, P. (2012) 'EASEA: specification and execution of evolutionary algorithms on GPGPU', *Soft Computing Journal*, Springer, Vol. 16, No. 2, pp.261–279.

Oiso, M., Matsumura, Y., Yasuda, T. and Ohkura, K. (2011) 'Implementing genetic algorithms to CUDA environment using data parallelization', *Technical Gazette*, Vol. 18, No. 4, pp.511–517.

Ouannes, N., Djedi, N., Luga, H. and Duthen, Y. (2012) 'Gait evolution for humanoid robot in a physically simulated environment', *Intelligent Computer Graphics Book*, Springer, pp.157–173.

Pai, S., Govindarajan, R. and Thazhuthaveetil, M. J. (2008) 'Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme', *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ACM, New York, pp.33–42.

Patel, R.A., Zhang, Y., Mak, J., Davidson, A. and Owens, J.D. (2012) 'Parallel lossless data compression on the GPU', in *Proceedings of IEEE Innovative Parallel Computing (InPar '12)*, San Jose, CA.

Patulea, C., Peace, R. and Green, J. (2014) 'CUDA-accelerated genetic feedforward-ANN training for data mining', *Journal of Physics: Conference Series*, IOP Publishing, Vol. 256, No. 1, pp.12–14.

Peniak, M., Morse, A., Larcombe, C. and Ramirez-Contla, S. (2011) 'Aquila: an open-source GPU-accelerated toolkit for cognitive and neuro-robotics research', *The International Joint Conference on Neural Networks (IJCNN '11)*, IEEE, pp.1753–1760.

Petrovic, P. (2004) *Distributed System for Evolutionary Robotics Experiments*, Technical Report 05/04, Department of Computer and Information Science, Norwegian University of Science and Technology.

Pospichal, P., Jaros, J. and Schwarz, J. (2010) 'Parallel genetic algorithm on the CUDA architecture', *Applications of Evolutionary Computation Book, LNCS*, Springer-Verlag, Berlin, Heidelberg, Vol. 6024, pp.442–451.

Prabhu, R.D. (2007) 'Gneuron: parallel neural networks with GPU', *International Conference on High Performance Computing (HiPC '07)*, Citeseer.

Qin, A.K., Raimondo, F., Forbes, F. and Ong, Y.S. (2012) 'An improved CUDA-based implementation of differential evolution on GPU', in Terence Soule (Ed.): *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO '12)*, ACM, NY, USA, pp.991–998.

Rubrecht, S., Singla, E., Padois, V., Bidaud, P. and De Broissia, M. (2011) 'Evolutionary design of a robotic manipulator for a highly constrained environment', *Computational Intelligence Journal*, Springer, Vol. 341, pp.109–121.

Salmon, J.K., Moraes, M.A., Dror, R.O. and Shaw, D.E. (2011) 'Parallel random numbers: as easy as 1, 2, 3', *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, ACM, New York, pp.1–12.

Satish, N., Harris, M. and Garland, M. (2009) 'Designing efficient sorting algorithms for many core GPUs', *IEEE International Symposium on Parallel & Distributed Processing, (IPDPS '09)*, IEEE, pp.1–10.

Scalabrin, M. H., Parpinelli, R.S., Benítez, C.M. and Lopes, H.S. (2014) 'Population-based harmony search using GPU applied to protein structure prediction', *International Journal of Computational Science and Engineering*, Inderscience, Vol. 9, No. 1, pp.106–118.

Shah, R., Narayanan, P. and Kothapalli, K. (2010) 'GPU-accelerated genetic algorithms', *Proceedings of the Workshop on Parallel Architectures for Bio-inspired Algorithms*, pp.27–34.

Shi, M., Pan, W., Garis, H. and Chen, K.(2010) 'Approach to controlling robot by artificial brain based on parallel evolutionary neural network', *2nd International Conference on Industrial Mechatronics and Automation (ICIMA '10)*, IEEE, Vol. 2, pp.502–505.

Skolicki, Z. (2005) 'An analysis of island models in evolutionary computation', *Proceedings of the 7th Annual Workshop on Genetic and Evolutionary Computation (GECCO '05)*, ACM, pp.386–389.

Sofge, D.A., Potter, M.A., Bugajska, M.D. and Schultz, A.C. (2003) 'Challenges and opportunities of evolutionary robotics', *Proceeding of the Second International Conference on Computational Intelligence, Robotics, and Autonomous Systems (CIRAS '03)*, Singapore.

Talbi, E. (2009) *Metaheuristics: From Design to Implementation*, Wiley Publishing, Vol. 74, p.624.

Todorov, E., Erez, T. and Tassa, T. (2012) 'MuJoCo: a physics engine for model-based control', *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp.5026–5033.

Torres, Y., G-Escribano, A. and Llano, D.R. (2012) 'Measuring the impact of configuration parameters in CUDA through benchmarking', *The 12th International Conference Computational and Mathematical Methods in Science and Engineering (CMMSE '12)*, Vol. 12, pp.33–47.

Wang, S., Chaovalitwongse, W. and Babuska, R. (2012) 'Machine learning algorithms in bipedal robot control', *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, IEEE, Vol. 42, No. 5, pp.728–743.

Yang, Y., Xiang, P., Kong, J., Mantor, M. and Zhou, H. (2012) ‘A unified optimizing compiler framework for different GPGPU architectures’, *ACM Transactions on Architecture and Code Optimization (TACO)*, ACM, Vol. 9, No. 2, pp.9:1–9:33.

Yeh, T.Y., Faloutsos, P. and Reinman, G. (2006) ‘Enabling real-time physics simulation in future interactive entertainment’, *ACM SIGGRAPH Symposium on Videogames*, ACM, New York, pp.71–81.

Zamith, M.P.M., Clua, E.W.G., Conci, A. and Montenegro, A., Leal-Toledo, R.C.P., Pagliosa, P.A. and Valente, L. (2008) ‘A game loop architecture for the GPU used as a math coprocessor in real-time applications’, *Computers in Entertainment (CIE)*, ACM, Vol. 6, No. 3, p.42.

Zhu, W. (2009) ‘A study of parallel evolution strategy: pattern search on a GPU computing platform’, *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation (GEC ‘09)*, ACM, New York, pp.765–772.