CUDA Massively Parallel Trajectory Evolution

M. Kashif Kaleem (King's College London), Jürgen Leitner[1] (IDSIA, University of Lugano)

I. Introduction

At the European Space Agency a software framework was developed over the last few years focusing on global optimization. The framework, mainly developed in-house at the Advanced Concepts Team (ACT), was released as an open-source project last year, that allows to run high dimensional global optimization problems, like interplanetary trajectory planning and optimizing controllers for space robotics. PaGMO is implemented in an object-oriented fashion as a C++ library but also has a Python wrapper for easy interactive sessions [1].

The core library includes support for multi-core systems (mainly through multi-threading) and recently has been trying to add functionality towards cluster and networked computers by aiming for an MPI integration. The project described in this paper was published as a possible Google Summer of Code project on the PaGMO sourceforge page but was dropped because of limited funding. Over the last year the aim was set to implement capabilities to run the PaGMO optimization heuristics on GPU architectures.

As a starting point the evolving docking problem was chosen, which aims to design a neuro-controller for the automatic rendezvous and docking task in spacecraft operations. A genetic algorithm is used to search for a neural network that controls the spacecraft [2].

The docking process involves a series of maneuvers and controlled thruster burns, which lead to the chaser vehicle eventually being able to come close and finally attach to the, in our case passive, target spacecraft. It can be seen as a variant of the phototaxis task in robotics, but using a different physical environment (no friction, non-zero external forces, thruster-based motion). The spacecraft is modeled as a circle with thrusters at the edge of its size (radius), which control forward/backward and rotational acceleration.

II. Design

Pagmo's design incorporates an island based Genetic Algorithm [3] framework for solving multiple objective optimization problems.

The structure of a PaGMO program includes a description of the problem to solve, an algorithm type and an archipelago. The problem description details how the genotype of an individual defines the fitness value of the individual. The algorithm defines how individuals in an island evolve and mutate while the archipelago describes the multi-island configuration of the genetic algorithm with migration etc.

The requirement of integrating GPGPU into PaGMO stems from the fact that some problem take an exuberant amount of time to solve. This is down to the number of evaluations and the complexity of the problem's fitness functions. These functions are ideal candidates for the SIMD type structure for GPGPU feasible algorithms. The challenge of the integration is in being able to simulate the genetic algorithm on the GPU while taking in consideration the different levels that exist in the multi-island genetic algorithm. Each PaGMO problem is originally solved by an archipelago of islands, each containing a number of individuals. We added two additional levels for points and task size.

Since the fitness function of each problem usually involves execution of some other sub-tasks (such as the simulation of neural networks which we are training), each individual can have a number of tasks within it. These are called points. Since each point itself can be computed in parallel (such as a feed-forward neural network where each neuron depends on its inputs independently), each point is assigned a task size.

The implementation breaks down the implementation into several classes.

A. cuda info

Provides details of the capabilities of the GPU device for instance dimensional limits and shared memory amount.

B. Kernel dimensions

Since the performance of the GPU code depends heavily on the choice of kernel dimensions it makes sense to have a set of classes to control this parameter. This depends on the amount of shared memory as well as the device's capabilities. These classes provide a means for specifying the kernel launch dimensions on which it enforces constraints to calculate the preferred dimensions.

C. dataset

This class encapsulates access to the GPU device memory. It provides memory accessors that have specifiers for island, individual and point id. The data is organized so that contagious threads access contagious data meaning that data for the same thread is organized along the y dimension.

D. task

The implementation revolves around the task class which describes the fundamental job that the GPU will be tasked to

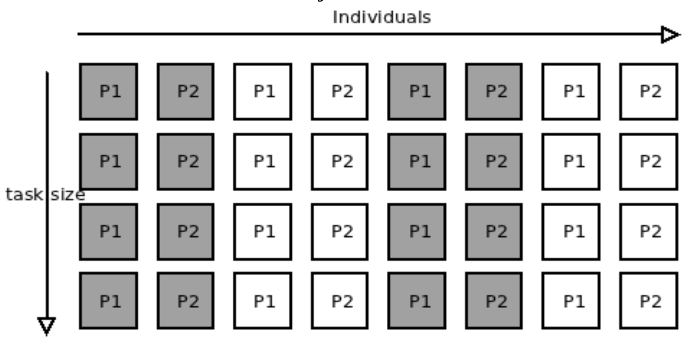

Fig. 1: This diagram shows the thread layout of an island containing 4 individuals, each with 2 points with task size of 4.

[1]The main contribution during this work was done while at the European Space Agency's Advanced Concepts Team.

complete. It can be used to launch more than one kernel if required. It also enables preconditions and postconditions as functors. To create a task, one must derive from this class and specify an implementation for the launch method. The launch method calls one or more CUDA kernels and passes device memory pointers to the actual CUDA kernel.

E. cuda problem

We aim to compute the fitness of the entire archipelago's islands and individuals in one go. We achieve this by defining a CUDA problem base class, which can be derived by implementing a method which computes the entire archipelago's fitnesses at the same time.

III. Docking problem

The problem of training neuro-controllers for the docking problem was broken down into three parts along similar lines to the original legacy implementation.

A. Neural network toolkit

A simplistic neural network toolkit was implemented that would be simulated on the GPU. The weights of the neural network are extracted from the individual's genome while the number of input sets form the number of points for the task. Two neural network types were implemented; simple and multilayer perceptrons.

B. Integrator class

Similar to the neural network toolkit, a set of integrators were implemented that took an equation as a device function type as a parameter and evaluated the ODE integral for that equation with additional parameters. The Runge-Kutta and Euler methods were implemented. In the case of the docking problem, the parameter function was an implementation of the Hill-Clohessey-Wiltshire (HCW) equations.

C. Fitness functions

To evaluate the fitness of the neural network being used by the docking problem, a set of fitness functions were implemented which would compare the output of the neural network with a reference value.

D. Docking problem

The docking problem used a multilayer perceptron that had 7 inputs, 11 neurons at the hidden layer and 2 output neurons. This results in a chromosome size of 112 real values. The most significant part of the docking problem's implementation is its objfun function which starts off by loading up the neural network's weights and inputs. Since the integrator's inputs are common to that of the neural network, we just use pointers of those parameter. After loading the data, we launch the tasks in sequence for the duration of the specified docking time and then compute and set the fitness values back to the individuals. We also compute the fittest individual at this point.

IV. Performance and results

To evaluate the performance of this implementation, we compared it with the previous implementation with different configurations. Figures 2 and 3 show the time consumed for the legacy system in comparison to the GPU enabled implementation. For the tests the following hardware was used: a 1.6 Ghz dual core processor, 2 GB RAM and a 256 MB Nvidia ION GPU, which is fairly a low-end GPU but it serves our purpose of drawing a comparison between the implementations.

V. Conclusion

The aim of this project was to implement a scalable framework for using CUDA to perform processor intensive tasks while adhering to the design specification of PaGMO. This project succeeded in doing this and produced significant speed-ups for the docking problem implementation, especially for larger configurations. This improvement can be further improved by the use of better hardware.

Acknowledgment

We would like to thank Dr Francesco Biscani at the Advanced Concepts Team of the European Space Agency for the advice and support he provided during this project.

References

[1] Biscani, F., Izzo, D., & Yam, C. H. (2010). A Global Optimisation Toolbox for Massively Parallel Engineering Optimisation. 4th International Conference on Astrodynamics Tools and Techniques .

[2] Jürgen Leitner, Christos Ampatzis, Dario Izzo. (2010). Evolving ANNs for Spacecraft Rendezvous and Docking. 10th International Symposium on Artificial Intelligence, Robotics and Automation in Space.

[3] Ampatzis, C., Izzo, D., Rucinski, M., and Biscani, F., (2009). ALife in the Galapagos: migration effects on neuro-controller design. The European Conference on Artificial Life.

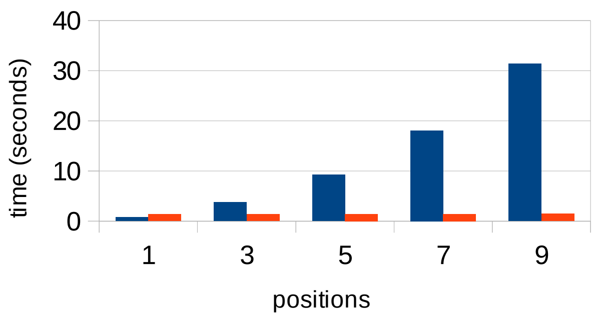

Fig. 2: This chart shows how the time consumed by the legacy implementation (blue) relates to the amount consumed by the CUDA implementation (orange) for varying number of positions.

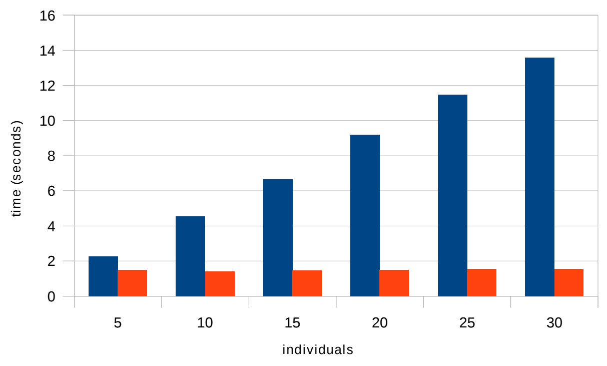

Fig. 3: This chart shows how the time consumed by the legacy implementation (blue) relates to the amount consumed by the CUDA implementation (orange) for varying number of individuals.