ON THE USE OF GPUS FOR MASSIVELY PARALLEL OPTIMIZATION OF LOW-THRUST TRAJECTORIES

Alexander Wittig, Viktor Wase, Dario Izzo

ESA Advanced Concepts Team, ESTEC, 2200AG Noordwijk, The Netherlands
{alexander.wittig, viktor.wase, dario.izzo}@esa.int

ABSTRACT

The optimization of low-thrust trajectories is a difficult task. While techniques such as Sims-Flanagan transcription give good results for short transfer arcs with at most a few revolutions, solving the low-thrust problem for orbits with large numbers of revolutions is much more difficult. Adding to the difficulty of the problem is that typically such orbits are formulated as a multi-objective optimization problem, providing a trade-off between fuel consumption and flight time.

In this work we propose to leverage the power of modern GPU processors to implement a massively parallel evolutionary optimization algorithm. Modern GPUs are capable of running thousands of computation threads in parallel, allowing for very efficient evaluation of the fitness function over a large population. A core component of this algorithm is a fast massively parallel numerical integrator capable of propagating thousands of initial conditions in parallel on the GPU.

Several evolutionary optimization algorithms are analyzed for their suitability for large population size. An example of how this technique can be applied to low-thrust optimization in the targeting of the Moon is given.

Index Terms— low thrust, global optimization, GPU, evolutionary, parallel, ODE integration

1. INTRODUCTION

Techniques such as Sims-Flanagan transcription are commonly employed in the optimization of low-thrust trajectories. They are fast and give good results for short transfer arcs with at most a few revolutions. However, solving the low-thrust problem for orbits with large numbers of revolutions is much more complex. Part of this complexity is that typically such trajectory optimization problems are formulated as a multi-objective problems, providing a trade-off between fuel consumption and flight time.

Traditional algorithms to solve this kind of problem have been developed considering the typical computing architecture of (potentially multicore) Central Processing Unit (CPU) machines. In these systems, a small number of computationally powerful cores with fast random memory access due to several layers of memory caches is available. With the advent of modern GPUs a huge number of Arithmetic Logic Units (ALU) per card (typically on the order of thousands) are made available at costs comparable to those of a single high-end multicore CPU [1]. However, the large increase in raw arithmetic computing power comes at the cost of limited performance of each core compared to traditional CPU cores mostly due to the lack of the sophisticated hardware caches present in CPUs.

This GPU architecture is not suitable to run different, complex subprograms on each core. Instead, the hardware is optimized for what is referred to as the Single Instruction Multiple Threads (SIMT) parallel processing paradigm [2], which applies the same operation to different input data in parallel. This requires a different approach in algorithm design to maximize the impact of those new hardware capabilities in various fields of scientific computing [3].

Early attempts have been made at using GPUs in global optimization of docking dynamics of spacecraft [4]. More recently, GPU programming has found its way into many other aerospace related research topics. GPUs have been used for the parallel computation of trajectories to enable fast Monte-Carlo simulations [5], uncertainty propagation [6], as well as non-linear filtering [7] and fast tree-search algorithms [8].

In this work we propose a massively parallel evolutionary optimization algorithm using the GPU to for massively parallel evaluation of the fitness function over a large population. This fitness function requires the propagation of a spacecraft state over many revolutions. Thus a core component of this algorithm is a numerical integrator implementation capable of propagating thousands of initial conditions in parallel on the GPU.

We chose the standardized OpenCL package [9] for the GPU programming. The OpenCL heterogeneous computing platform provides a platform agnostic C like programming language along with compilers for most current accelerator cards such as those by NVidia, Intel and ATI. Furthermore, it provides libraries for both the host side (CPU) as well as the client side (GPU) to facilitate common tasks in heterogeneous programming in a hardware independent manner. Interfacing the OpenCL code for GPU fitness function evaluation with the Python language allows for an easy to use interface to the otherwise somewhat cumbersome C code while maintaining the high performance required in the critical code paths.

We consider several evolutionary optimization algorithms for their suitability for large population size. In order to be implemented efficiently on a GPU, it is necessary for the algorithms to have a high level of concurrency and a complexity scaling well with the population size.

Lastly, we show an example of how this implementation can be applied to low-thrust optimization in the targeting of celestial bodies, such as the Moon. While we perform the propagation in a simple two-body model, due to its numerical nature in principle the propagation can also be performed in more complete models such as the circular restricted three body problem.

The remainder of this paper is structured as follows. In Section 2 we introduce the GPU based Runge-Kutta integrator implementation at the core of the fitness function. We then proceed to give an overview of various implementations of evolutionary optimizers in Section 3. Following this we illustrate the concept by optimizing a simple low thrust example in Section 4. We finish with some conclusions in Section 5.

2. GPU INTEGRATOR

In principle, ODE integration is an inherently sequential process. The propagation of a single state almost never benefits from the implementation of a GPU. This is because in virtually all integration schemes a single step corresponds to the evaluation of the right hand side followed by the calculation of a new intermediate state which is used to evaluate the right hand side again.

The highly parallel execution on a GPU comes into play when propagating a set of initial conditions instead of s single state. In that case, all operations of the integrator can be performed on all initial states at once, propagating at the same time the full set of initial conditions.

2.1. Implementation

We chose to implement an arbitrary Runge-Kutta integration scheme with automatic stepsize control such as the Runge-Kutta-Fehlberg 4/5 [10] or a corresponding 7/8 order method [11]. Such implementations have been described in the literature before [12, Chapters 7,8]. However, we found previous implementations not very well adapted to the computation paradigm of GPUs, leading to poor relative performance when compared with CPU based implementations. The implementation in [12, Chapters 8], for example, finds only a speedup of a factor of about two when compared with parallel execution on a quad-core CPU. Our implementation, instead, is carefully tuned to GPUs, and tested extensively in particular on the AMD Graphics Core Next (GCN) architecture [2].

One particularly important fact in this context is the well known observation that branching of any sort in GPU code is very costly and should be avoided [13]. In a naive implementation of RK integration schemes with step size control, many different conditions are constantly checked to determine if the solution is within the user specified error bounds, step-size limits and propagation time bounds are met, etc. Also loops result in potentially costly branching instructions if the compiler cannot automatically unroll them.

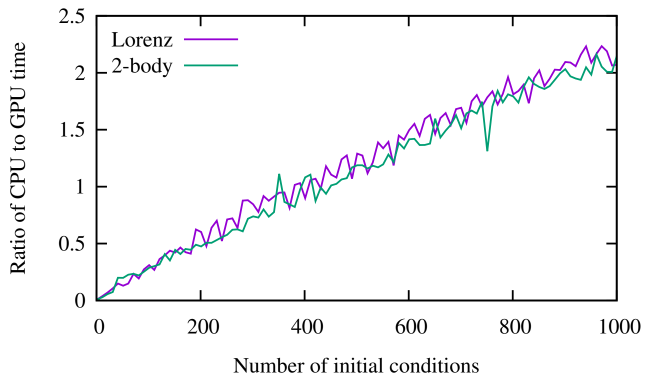
We avoided these pitfalls by optimizing our code to use specific GPU commands instead. Of particular use in this context was the conditional assignment operator (called select() in OpenCL), which selects between two values based on the value of a third parameter. On GPUs this operator is implemented in the instruction set of the processor and hence very fast. It does not require any branching, and can replace many instances of if statements. Further optimizations were achieved by manually unrolling loops in the code. While the AMD OpenCL compiler has flags to force it to unroll loops automatically, we found that some loops that could be unrolled manually were not automatically unrolled by the compiler.

With these optimizations, we were able to replace all if statements in the code and all but one loop. This has been verified by checking the resulting machine code on the AMD GCN Hawaii platform (used on our AMD W8100 card) using the AMD Code XL profiling tool [14]. The speedup due to these optimizations was impressive, on the order of a factor of 5 to 10 for simple ODEs such as the Lorenz equation.

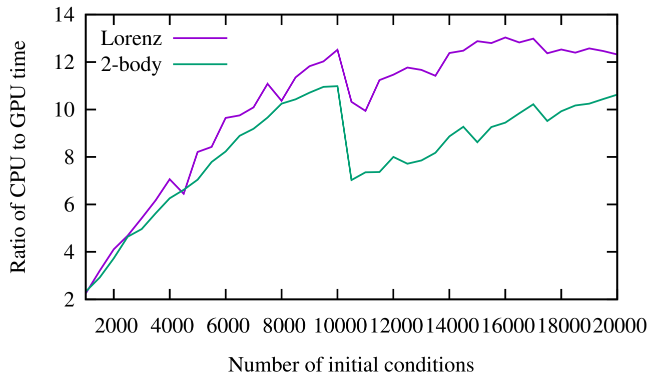
2.2. Performance

In the following, we perform some simple comparisons between our GPU integrator and CPU based integration for large numbers of initial conditions. The comparisons are carried out on a machine with a 4 core (8 hyperthreads) Xeon E5-1620 V3 at 3.5 GHz with 16 GB of RAM and an AMD Fire-Pro W8100 (2560 stream processors) at 850 MHz with 4 GB RAM. All computations are preformed using double precision floating point numbers, which in the latest generation of high-end GPUs are now well supported also on those devices at high speed.

In order to compare the performance of the integrator between CPUs and GPUs, we used a feature of the OpenCL environment. By simply switching the computation platform, it is possible to run the exact same code once on the CPU and once on the GPU. In case of the CPU the code will automatically make use of all available cores as well as the same auto-vectorization features used to generate the GPU code. This allows for a reasonable comparison between the two platforms without having to recode the entire test case. The AMDAPP OpenCL environment we used on our Windows machine uses a modified version of the LLVM based Clang compiler to compile the OpenCL code for the selected platform. Monitoring system load during execution confirms that all 8 available logical cores (Intel quad-core with hyper-threading) of



(a)



(b)

Fig. 1. Ratio of CPU execution time over GPU execution time as a function of number of initial conditions for different ODEs.

the CPU are fully loaded during CPU execution.

The first test is the simple three dimensional Lorenz ODE. This ODE has a particularly simple right hand side which is not very computationally intensive to evaluate. This ODE mostly tests the overhead introduced by the integrator implementation as those operations largely outnumber the computational effort in the evaluation of the right hand side of the ODE. More relevant for the application to low thrust propulsion is the integration of the spacecraft motion in two-body dynamics. In the second test case, we integrate instead the slightly more complex two body dynamics (in Cartesian coordinates) with constant tangential thrust. Here both the dimensionality as well as the computational complexity of the right hand side are increased.

In Figure 1 we compare the ratio of the computational times for parallel integrations performed on the CPU and the GPU in both test cases as a function of the number of initial conditions. In all cases, the exact same initial and final conditions are being used. As expected, for low numbers of initial conditions (Figure 1 (a)), the CPU is significantly faster due to its 4 times faster clock speed and 8 logical cores as well as lower overhead in setting up the computation. Considering, for example, the case of the Lorenz system, in the extreme case of just one single initial condition, the CPU is about 100 times faster than the GPU (1.25 ms compared to 128 ms). However, already at 30 initial conditions the GPU is only 10 times slower (13.7 ms vs. 129 ms). At about 400 initial conditions, the GPU comes in about even with the CPU. For initial condition sizes past that, the GPU starts to beat the GPU. As seen in Figure 1 (b), the linear trend actually continues to numbers of initial conditions around 10000, from where on out the GPU remains about 12.5 times faster than the CPU.

The situation for the propagation of a spacecraft state in the two-body problem looks similar. In fact for low numbers of parallel initial conditions up to about 1000, the trend is the same (Figure 1 (a)). Also for large numbers of initial conditions (Figure 1 (b)) the behavior is qualitatively the same. The maximum performance gain compared to CPU computations in this case is about 11, reached at 10000 and 20000 initial conditions.

It is interesting to observe that at 10000 initial conditions there is a significant drop in the speed-up for both ODEs. After the drop, the performance then keeps increasing again. We attribute this drop in performance to the fact that our particular GPU has 40 compute units, each of which is equipped with 4 SIMD units which are scheduled with a wavefront of 64 threads each. At full occupation this yields a maximum number of $40 \cdot 4 \cdot 64 = 10240$ computations scheduled in parallel. As our problem is processor limited, the most efficient use of GPU resources occurs at full GPU occupation.

These results indicate that the parallel propagation of spacecraft states on the GPU should be possible and yield a sizeable speedup over comparable integration performed on a serial CPU. The maximum performance increase on our particular GPU is reached when the number of initial conditions is about 10000, but already at much lower numbers of initial conditions, such as 1000, is it possible to benefit noticeably from GPU computation.

3. OPTIMIZATION ALGORITHMS

In the following Section, we investigate various algorithms for their suitability for large population sizes as well as other aspects that affect their performance in a massively parallel environment.

One common feature of almost all of these algorithms is that typical implementations of these algorithm are steady-state, i.e. the mutations are performed sequentially and any changes to the population immediately take effect before continuing with the next individual. In order to parallelize these algorithms, it is necessary to make the algorithm generational. In that way all mutations can be evaluated in parallel, yielding a new population in one step. In general, this makes the algorithms less performant as improvements during the mutations only feed back into the algorithm after one generation.

3.1. Algorithms

We considered the following algorithms. The presentation of the algorithms is kept very brief, for details about each algorithm the reader is referred to the scientific literature in the references.

3.1.1. *Particle Swarm Optimization Algorithm*

The Particle Swarm Optimization Algorithm is an optimization method based on swarm intelligence. Each particle has a position vector X_i and velocity vector V_i as well as two stored positions P_i and S_i. The first variable P_i the position that has the greatest fitness out of all the positions particle i has visited. The variable S_i is usually the best position the swarm has visited, but in order to reduce the chance of convergence in a local minimum each particle is only allowed to communicate with a few other particles. In this case a ring topology is used. The S_i variable is thus the best position of the particle i or its nearest neighbours.

A canonical Particle Swarm Optimizer implementation was used, thus giving an update equation of

$$V_i^{t+1} = \alpha\left(V_i^t + \phi_1 U_1(P_i - X_i^t) + \phi_2 U_2(S_i - X_i^t)\right)$$

where

$$\alpha = \frac{2k}{|2 - (\phi_1 + \phi_2) - \sqrt{(\phi_1 + \phi_2)^2 + 4(\phi_1 + \phi_2)}|}$$

with the values $\phi_1 = \phi_2 = 2.05$ and $k = 1$. Both U_1 and U_2 denotes vectors drawn from a uniformly random distribution between 0 and 1. Note that a maximum velocity of $0.5L$ is imposed, where L is the length of the bounding box of the problem.

The position of each particle is then updated as

$$X_i^{t+1} = X_i^t + V_i^{t+1}$$

3.1.2. *Simple Genetic Algorithm (SGA)*

There are plenty of variations of genetic algorithms. One of the more common, and the one we consider here, seems to be the variant with roulette wheel selection, elitism and exponential crossover.

First, the fitness of each of the individuals in the population is evaluated. To create the population of the next generation, for each new individual 2 parents are chosen at random from the old population with the probability of a parent being picked being proportional to its fitness. This is referred to as roulette wheel selection.

In order to combine the two parents, an exponential crossover is used. This means that the genes are copied from one of the parents to the child, one by one, but with each gene there is a p_{cr} probability that the parent from whom the genes are copied is exchanged for the other parent. We use a value of $p_{cr} = 0.05$.

In the next step, each of the newly minted individuals are mutated. This means that for each of the genes there is a p_{mr} chance that the gene is mutated, where $p_{mr} = 0.02$ is a tunable parameter. If a mutation takes place, $c_{mr}\epsilon$ is added to the value of the gene, where $c_{mr} = 0.1$ is another tunable parameter and ϵ is drawn from a normal distribution.

Lastly, elitism means that the best N_{el} individuals from the previous generation are always passed on, un-changed, to the next generation.

3.1.3. *Self-Adaptive Differential Evolution (JDE)*

The classical differential evolution algorithm takes 2 parameters p and f. For each new generation, the individuals of the existing population are updated according to the following rules: First, three random, but distinct, individuals A, B, C are selected from the previous generation. A new individual Y is created from these three individuals according to $Y = A + f(B - C)$ where f is a tunable parameter. Such a Y will be created for each individual X in the population, and these will crossover in order to create a new trial individual Z. In this case an exponential crossover is applied, which means that a random point q is chosen uniformly such that $0 \leqslant q < G$ and $q \in \mathbb{N}$, where G is the number of genes. All genes from q to the end-point of the crossover q_e are copied from Y into Z. All genes after the end point are copied from X into Z. The endpoint q_e is defined as $q + \eta \mod D$, where η is a random integer drawn from a geometric distribution with intensity parameter p. If the fitness of Z is greater than that of X, Z replaces X.

In JDE each individual contains the original gene as well as values for f and p. Before computing the next generation there is a 10% chance that either of them are mutated. During mutation, f_i is drawn randomly from a uniform distribution on $[0.1, 1]$ while p_i is drawn from a uniform distribution on $[0, 1]$. These new values are then used in the computation of the next generation.

3.2. Analysis

Denoting by N the size of the population, T the number of generations and G the length of the gene, all of these algorithms have a complexity of $O(NGT)$. Since this is linear in N, at least on the algorithm level they are well suited for large population sizes.

To analyze more closely the dependence of the population size on the convergence of each algorithm, we selected four commonly used test problems, Ackley, Rastrigin, Rosenbrock, and Schwefel, at 15 dimensions. For each of these, we define a convergence limit for the solution to be considered converged. Each algorithm (using the implementation in PaGMO [15]) is run 10 times. We then calculate the average number of generations needed to obtain a convergent solution. If the solution does not converge after $15,000$ generations, the

Ackley Problem

	Population Size = 10	Population Size = 100	Population Size = 500
SGA	20350	10620	10010
JDE	2216.7	1850	1780
PSO	2442.9	2080	2010

Rastrigin Problem

	Population Size = 10	Population Size = 100	Population Size = 500
SGA	16140	7090	7480
JDE	3325	2810	2760
PSO	-	-	-

Rosenbrock Problem

	Population Size = 10	Population Size = 100	Population Size = 500
SGA	6800	8350	10887.5
JDE	2330	1770	1660
PSO	4760	1420	1160

Schwefel Problem

	Population Size = 10	Population Size = 100	Population Size = 500
SGA	-	-	3925
JDE	2550	1950	1880
PSO	-	-	-

Table 1. Mean number of generations required until the convergence criteria of the respective problem is met. A dash indicates that none of the runs converged.

method is considered non-convergent and is not counted for the averaging.

Note that at this point we are not yet considering the computational cost of increasing the population size. Instead, we simply want to determine if a larger population size, assuming for now that it comes at zero additional cost, is advantageous for the algorithm.

The result is shown in Table 1. As can be seen, the number of generations required to arrive within the specified tolerance of the solution generally decreases with the population size. However, the difference between 100 and 500 individuals is not very pronounced, indicating that in our test cases an increase in the population size generally does not translate into faster convergence.

A surprising exception seems to be the genetic algorithm (GA), which actually performs worse with larger population size for the Rosenbrock test function. This is quite counterintuitive as theoretically the probability of finding the optimum increases with the population size. At the moment we cannot explain this behavior.

From this analysis we decided to select the JDE algorithm as the candidate for our GPU based optimization. Furthermore, it appears from these simple tests that, at least for single objective optimization, a larger population size is not necessarily beneficial for convergence speed. However, in multi-objective optimization more individuals allow a better coverage of the Pareto front. Another area where large populations are relevant in practice are island models, in which many populations are evolved in parallel with occasional migrations between the islands [16].

4. EXAMPLE

Just to illustrate the feasibility of the proposed method, we implemented very simple dynamics of a spacecraft with low thrust propulsion. The dynamics are formulated in Cartesian coordinates, with a fixed thrust direction along the tangential direction. The thrust magnitude is governed by a 20th order Bernstein polynomial $P(f)$ as a function of the true anomaly f. The thrust from the polynomial $P(f)$ is then truncated to the interval $[0, T_{max}]$ where T_{max} is the maximum thrust.

A spacecraft with a low thrust propulsion engine with maximum thrust of $T_{max} = 0.1$ N is initially in a circular LEO orbit at $6,778.14$ km. With an initial chemical impulse of 3 km/s the spacecraft then has to spiral out to the Moon's orbit (assumed as circular at $380,000$ km) as quickly as possible. Since the direction and time of the initial impulse can be chosen arbitrarily, we can ignore the phasing of the Moon and the spacecraft for this very simple application.

The optimization is performed on the time of flight required to reach a position $380,000$ km from Earth. The optimization parameters are the Bernstein coefficients a_i of the thrust polynomial P. The optimization algorithm we used is JDE with a population size of 400 individuals computed on the GPU.

This is a rather simple problem to optimize and hence the optimizer converges fairly quickly to a solution. Figure 2 shows the evolution of the orbit and the thrust profile. The best solution after 6 generations has a flight time of 359.7 days.

5. CONCLUSIONS

We demonstrated that efficient GPU implementation of an efficient numerical ODE integrator allows for typically computationally intensive fitness functions, such as those requiring numerical integration, to benefit significantly from massively parallel GPU execution. This enables the parallel evaluation of the fitness of thousands of individuals in parallel with minimal computational cost. These new possibilities have the potential to provide significant speed-ups compared to sequential CPU implementations.

The feasibility of the proposed method is illustrated by a very simple example, showing convergence of the JDE algorithm implemented with parallelized fitness evaluation on the GPU.

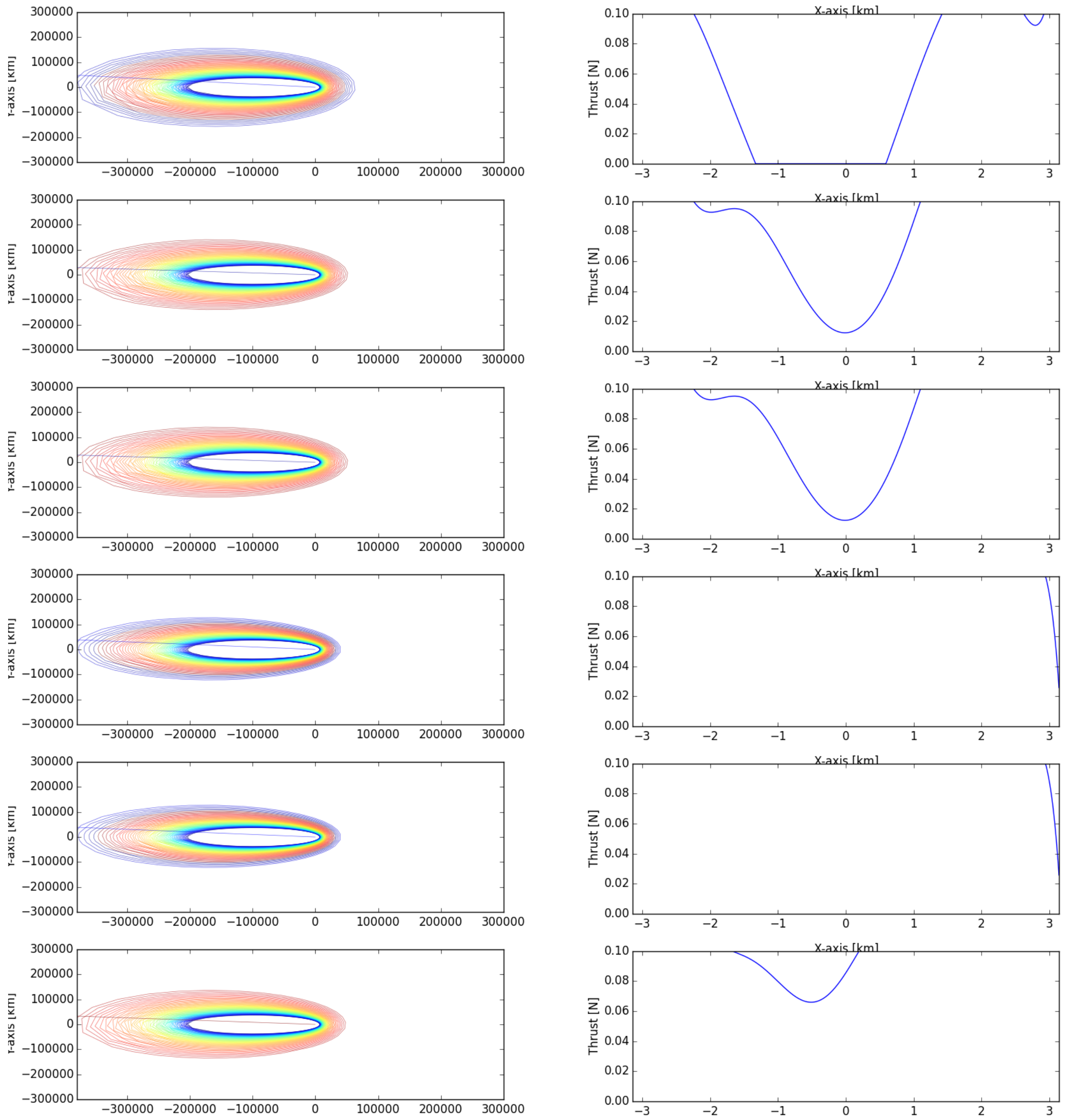


Fig. 2. Evolution of the best orbit over 6 generations (from top to bottom).

For classical single-objective optimization of some common test problems with commonly used evolutionary algorithms evidence suggests that larger population size does not automatically improve convergence. However, it may be possible to . Furthermore, in multi-objective problems it is expected that larger population size yields better quality results by providing increased coverage of the pareto front of the solution.

Further research is therefore required to identify evolutionary algorithms that can benefit from large population sizes. This may include different algorithms or techniques such as parallel execution of several instances of a particular algorithm with successive migration as in the case of an island structure in an archipelago.

6. REFERENCES

[1] J. Nickolls and W.J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, March 2010.

[2] Advanced Micro Devices (AMD), *AMD APP SDK OpenCL User Guide*, 1.0 edition, August 2015.

[3] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," 2007.

[4] M. Kashif Kaleem and Jürgen Leitner, "Cuda massively parallel trajectory evolution," in *GECCO*, 2011.

[5] Nitin Arora, Vivek Vittaldev, and Ryan P. Russell, "Parallel computation of trajectories using graphics processing units and interpolated gravity models," *Journal of Guidance, Control, and Dynamics*, pp. 1–11, June 2015.

[6] N. Nakhjiri and B. F. Villac, "An algorithm for trajectory propagation and uncertainty mapping on gpu," in *23rd AAS/AIAA Space Flight Mechanics Meeting*. American Astronomical Society, 2013, 2013-376.

[7] H. Shen, V. Vittaldev, C. D. Karlgaard, R. P. Russell, and E. Pellegrini, "Parallelized sigma point and particle filters for navigation problems," in *36th Annual AAS Guidance and Control Conference*. American Astronomical Society, 2013, 2013-034.

[8] Mauro Massari and Alexander Wittig, "Optimization of multiple-rendezvous low-thrust missions on general-purpose graphics processing units," *Journal of Aerospace Information Systems*, pp. 1 – 13, Jan 2016.

[9] Khronos Group, "Opencl The open standard for parallel programming of heterogeneous systems https://www.khronos.org/opencl/," .

[10] E. Fehlberg, "Classical fourth- and lower order runge-kutta formulas with stepsize control and their application to heat transfer problems," *Computing*, vol. 6, no. 1, pp. 61–71, 1970.

[11] J. H. Verner, "Numerically optimal runge–kutta pairs with interpolants," *Numerical Algorithms*, vol. 53, no. 2, pp. 383–396, 2009.

[12] Volodymyr Kindratenko, Ed., *Numerical Computations with GPUs*, Springer Berlin Heidelberg, 2014.

[13] Advanced Micro Devices (AMD), *AMD APP SDK OpenCL Optimization Guide*, 1.0 edition, August 2015.

[14] AMD Developer Tools Team, *AMD CodeXL Quick Start Guide*, Advanced Micro Devices (AMD), 1.8 edition, 2015.

[15] Dario Izzo, "Pygmo and pykep: open source tools for massively parallel optimization in astrodynamics (the case of interplanetary trajectory optimization)," in *Proceed. Fifth International Conf. Astrodynam. Tools and Techniques, ICATT*, 2012.

[16] Christos Ampatzis, Dario Izzo, Marek Ruciński, and Francesco Biscani, *Advances in Artificial Life. Darwin Meets von Neumann: 10th European Conference, ECAL 2009, Budapest, Hungary, September 13-16, 2009, Revised Selected Papers, Part I*, chapter ALife in the Galapagos: Migration Effects on Neuro-Controller Design, pp. 197–204, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.