Orbital interference, planetary close-approaches detection and memory handling on GPUs

Laurea Magistrale in Space Engineering - Ingegneria Spaziale

Author: Adriano Filippo Inno

Advisor: Prof. Camilla Colombo

Co-advisor: Alessandro Masat

Technical advisor: Lorenzo Bucci

Academic year: 2020-2021

1. Introduction

Graphics Processing Units (GPU)s provide much higher instruction throughput and memory bandwidth than a Central Processing Unit (CPU) within a similar price range [4]. High-efficiency GPU software can exploit this benefit and grant a huge computational time speed-up with respect to classical CPU programs, up to two order of magnitude. Recently, the employment of GPUs for space mission design-related programs is increasing. Orbit propagation software are suited to be accelerated by graphic cards, allowing to propagate tens of thousands bodies in parallel. However, the difficulties related to implementing efficient GPU algorithms for orbital interference and event detection during the propagation make the usage of graphic-cards less appealing. Moreover, GPU programming can be hard, and it is not the focus of space engineers. GPU software are particularly powerful in large-scale analyses, such as Monte-Carlo methods or grid search optimisations. However, handling the consequent huge sets of output arrays complicates the development of the software itself.

The aim of the work is to tackle the described problems suggesting some event detections algorithms that exploit logical arithmetic to efficiently run in a GPU based application, as well as efficient techniques to handle the outputs.
The work presents the development of the event detections algorithms in CUDAjectory, an open source orbit propagation GPU software available under European Space Agency (ESA) Community License, recently developed by the mission analysis team of ESA. The developments focus on the implementation of four algorithms: close-approach, Low Earth Orbit (LEO) protected region, Geostationary Earth Orbit (GEO) protected region, and massless bodies collision. A significant part of the work is also devoted to lower the computational effort of the output handling, which is the main bottleneck of CUDAjectory. The ultimate goal for the software is to enrich its interference and event detection capabilities in order to increase both the amount of analyses perform-able and the user pool.

2. Brief overview of CUDAjectory

CUDAjectory is written in C++, CUDA and Python. It is optimised to propagate millions of bodies, called *samples*, on GPUs. The output consists in the final states of the samples and the eventual events detected during the propagation. The user can exploit multiple gravity models and orbital perturbations to enrich the accuracy of the simulation.
The propagation of each sample is carried on with the patched conic approach [1]. An event algorithm detects the eventual change of Sphere Of Influence (SOI) and updates the integration centre of each sample. Another algorithm detects the collisions between the samples and the celestial bodies. The third event detection algorithm available in the previous version of CUDAjectory is called *close-approach*. Its optimisation is part of this work. The goal and the development of the algorithm is presented in Section 3. The position and velocity vector of celestial bodies is retrieved at running time by exploiting planetary ephemerides. The ephemerides data are an input of the simulation and must be passed using the NAIF-JPL Satellite and Planet Kernels (SPK) format. Many SPK types exist, and CUDAjectory is currently compatible with respect to SPK types 2 and 3 only. These two file formats are optimised to evaluate planetary motions.
In CUDA, the main functions executed on GPUs are called *kernels*, and are issued by the CPU. The samples are propagated in parallel by the computational units of the GPUs. In CUDAjectory, the simulation is divided in multiple sequential kernels, each one performing N consecutive integrational steps, where the amount of consecutive steps can be tuned by the user. This simulation layout comes form a previous optimisation of the ephemerides evaluation described in details in [2, 5]. After the execution of each kernel, the partial output containing the eventual events detected and the intermediate states of the samples is post-processed and the useful data are saved. In the previous version of CUDAjectory, the data related to the events are stored in multiple structures of doubles. Each event algorithm stores a specific amount of variables per detection, leading to additional post-processes to parse the partial output.

3. Close-approach algorithm

The goal of the close-approach algorithm is to detect the nearest point of each sample with respect to its closest celestial body during the propagation. The outcomes of the algorithm can be utilised by the users for multiple purposes, for example to check and identify potentials fly-by happened during the propagation. The previous version of this algorithm exploits a bisection-like procedure to converge at the event condition. The iterations are performed imposing the next time step according to the *range rate* (rr), which is the radial component of the relative velocity between a sample and its closest celestial body:

$$rr = \frac{\boldsymbol{r}_k^{rel} \cdot \boldsymbol{v}_k^{rel}}{||\boldsymbol{r}_k^{rel}||} \tag{1}$$

where the subscript k indicates the closest celestial body. When the range rate is equal to zero, the sample is in its close-approach. The iterations start when the algorithm detects a change of sign of the range rate; the event is contained in the time interval between that epoch (t_b) and the epoch at previous integration step (t_a). Fig. 1 shows the timeline during the propagation of a sample, together with its sign of the range rate function, supposing that the integration is carried on for 100 steps.

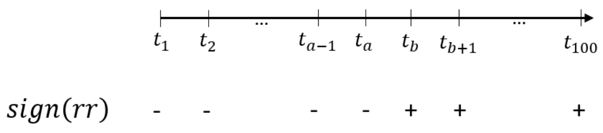

Figure 1: Timeline of the close-approach event during the propagation of a sample.

The rationale of the loop is to half the time interval at every iteration taking the part in which the event is contained, until a tolerance is met, like in a bisection. The loop is designed to enforce the sample to be after the event epoch at the last iteration, otherwise during the following integrational step the algorithm is triggered again by the change of sign of the range rate, leading to an infinite loop. In order to avoid this infinite loop condition, the time interval is not halved if the range rate is negative, imposing the next time step to the current upper boundary of the interval. The resulting conver-

gence rate is up to 30% lower than a classical bisection scheme. Moreover, the high number of if-else statements of previous implementation causes warp divergence (i.e. a typical GPU performance issue in which threads are stalled because of branch-dependant instructions) [4].
The work presents a new developed algorithm, aimed at achieving an higher convergence rate and reducing the warp divergence. The algorithm exploits a bisection-like loop as in the previous implementation, but halving the time interval at each iteration. The infinite loop condition is avoided by storing a dummy sign of the range rate and enforcing the epoch after the final iteration to be higher than the event epoch. The aim is to avoid any iteration in which the interval is not halved, in order to improve the performance. Logical arithmetic is used to keep the number of if-else statements as low as possible, in order reduce the warp divergence.

4. Earth protected regions algorithms

The goal of the LEO and GEO protected region algorithms is to detect the conditions at which the samples cross one of the boundaries of the two regions. This implementation aims to extend the simulation capabilities to Earth planetary protection and space debris analyses. Fig. 2 shows the conventions adopted to define the protected regions, according to the Inter-Agency space Debris coordination Committee (IADC) space debris mitigation guidelines [3].

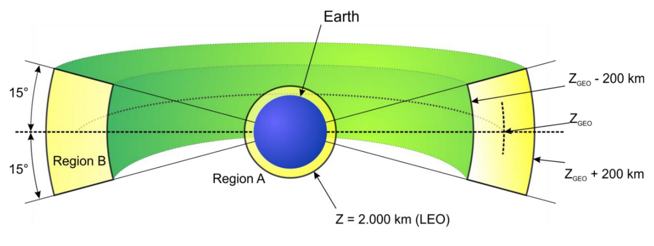

Figure 2: Earth protected regions - IADC conventions [3]

The thesis presents the reasons why a bisection-like method is not a robust choice for these two algorithms. Instead, an adaptive step refinement procedure is used to update the next time step according to a prediction of the altitude and the latitude (for the GEO algorithm only). Calling the altitude z, the latitude l, and the time step interval h, the predictions are:

$$\begin{aligned} z_{n+1} &\approx z_n + \dot{z}_n h_{n+1} \\ l_{n+1} &\approx l_n + \dot{l}_n h_{n+1} \end{aligned} \tag{2}$$

where indicates n the current time step. Eq. (2) are based on a linear extrapolation of the first order derivatives. The derivative of the altitude is given by the radial velocity of the sample, while the derivative of the latitude is approximated with the backwards finite difference of the latitude between two time steps. The predictions in Eq. (2) depend on the next time step interval (h_{n+1}) which is not known in advance. However, the work proposes a robust way to predict its worst possible value related to the integrator scheme used in the software. The predictions are used to identify potential crossings over the LEO or GEO boundaries at running time. If this happens, the step is adjusted in order to converge at the predicted crossed boundary.
Boolean algebra is exploited to minimise the warp divergence as in the close-approach algorithm.

5. Massless bodies collision algorithm

The Massless bodies collision algorithm is aimed at finding the collisions between the samples and a set of user-defined bodies. The radius of the collision sphere around each body is defined by the user. The massless bodies are modeled as orbiting points without a proper mass. Consequently, the gravity acceleration acting on the samples is not perturbed by this set of bodies and the integrational step-size is not intrinsically refined by the integrator. The massless bodies are not propagated by CUDAjectory. The goal is to utilise a user-prompted ephemerides file to evaluate the position and velocity of the requested massless bodies.
As presented in Section 2, the current compatibility of CUDAjectory with respect to SPK types limits the possibility to provide data for non-planetary bodies. For this reason, the thesis focuses on the analytical implementation of the Libration points motion, in order to be utilised as massless bodies. Libration points are equilibrium points under the influence of a primary and a secondary mass [1]. The proposed analytical implementation exploits the instantaneous

position and velocity vectors of the secondary body with respect to the primary to evaluate the motion of the Libration points using a roto-pulsating restricted three body problem model. The work presents the development of the massless bodies collision algorithm in analogy to the LEO protected region one. The proposed algorithm exploits an adaptive step refinement procedure based on the prediction of the distance (d) between each sample and a massless body, to converge at the detection. The prediction is computed as:

$$d_{n+1} \approx d_n + \dot{d}_n h_{n+1}$$

The derivative ($\dot{d}_n$) is given by the range rate, computed as in Eq. (1).
Since the Libration points are modelled as massless points, the time steps are not automatically refined in the proximity of the points themselves during the integration. As a consequence, the linear first order extrapolation is not robust enough. In order to improve the robustness of the prediction, the algorithm uses the magnitude of the sample velocity instead of the range rate, which represents an upper boundary of the range rate itself.
The warp divergence is minimised by exploiting logical computations as in the other algorithms.

6. Software optimisation

The output handling of the previous version of CUDAjectory takes more than the 99% of the total computational time in most of the simulations, representing a huge execution bottleneck. This part of the thesis is aimed at optimising the computational time of the output handling. The new implementation is based on the usage of a common data-type structure to contain the kernel output of every event detection algorithms, improving the final results retrieval efficiency. Since the detections are mutually exclusive, a single common data structure is allocated for each time step and for each sample. The size of the whole allocation can be in the order of tens of GigaBytes (GB). As a consequence, simulations are often limited by the memory capacity of the computer and the user must decrease the number of samples to be propagated. For this reason, the thesis focuses on the minimisation process of the common data structure size, performed to reduce the total memory required. Nowadays, operating systems deal with the so called *virtual memory* to store the applications data. The required memory is divided in small chunks of few KiloBytes (KB), called *pages*. When the required memory is higher than the physical memory available, the operating system relocates some pages from the physical memory into additional virtual pages. The additional pages are temporarily stored in the computer storage, causing a huge performance bottleneck in many applications. The new output handler ensures that the physical memory is enough to store all the required data, without using additional virtual pages and consequently avoiding potential bottlenecks.
CPU memory allocations are *pageable* by default, which means that can be moved into the storage if needed. It is possible to request physical, non-pageable memory, which is called *pinned* or *page-locked* memory. The work exploits the mapped pinned memory, which is a specific type of pinned memory, to allocate the kernel output and to further improve the performance.

7. Conclusions

The first part of the work proves that is possible to efficiently design GPU event detection algorithms, despite the difficulties of avoiding branch-dependent equations. Moreover, it demonstrates the validity of adaptive step refinement procedures to adjust the integrational time steps according to linear extrapolation predictions. The algorithms were successfully validated using Godot, a CPU-based software package for orbit propagation and optimisation available under ESA Community Licence.
The new output handler grants a massive computational time decrease, leading to analyses up to fifty time faster than before. Moreover, in the new version of CUDAjectory, analyses involving the close-approach algorithm benefit of an additional performance uplift up to 15%, depending on the configuration.

References

[1] Howard D Curtis. *Orbital Mechanics for Engineering Students.* Elsevier, The Boulevard, Langford Lane, Kidlington, Oxford, 2010.

[2] Márton Geda. Massive parallelization of trajectory propagations using gpus. Master's thesis, Delft University of Technology, 1 2019.

[3] IADC. Space debris mitigation guidelines, 7 2021.

[4] Nvidia. *CUDA C++ Programming Guide*, 11 edition, 6 2021.

[5] Fabian Schrammel. Alternative ephemeris representations for astrodynamical simulations on accelerators. Master's thesis, Technisch Univeritat Darmstadt, 1 2019.