OpenCL-Accelerated Simplified General Perturbations 4 Algorithm

Juan Andrés Fraire[1], Pablo Ferreyra[1], and Carlos Marques[1]

Universidad Nacional de Córdoba, Argentina
juanfraire@famaf.unc.edu.ar

Abstract. The number of space objects such as satellites, spacecraft, and debris are increasing significantly, and so is the need for tracking them for security and collision avoidance purposes. In this context, as parallelism is becoming a new paradigm, the need of implementing high performance propagators remain unmet.
For this, we implemented Simplified General Perturbations No. 4 (SGP4), a popular analytical orbital propagator, in OpenCL. OpenCL is a rising high performance and heterogeneous computation paradigm aimed to take the best of the processing elements on a given platform, in a parallel fashion, regardless of the underlying architecture.
Despite some considerations had to be taken, we prove that our development shows no significant calculation differences, while not only being hardware independent, but also boosting the performance notably by two orders of magnitude in several scenarios.

1 Introduction

In general, given an object position and velocity we can accurately measure their future status using mathematical models assuming no unexpected change in the way. The same applies to orbiting objects in space, such as satellites, spacecrafts, or debris; if we know their position and velocity, we can make a reasonable prediction on where that object will be. The more disturbances or perturbations the model include in the calculations, the better the precision in the result, and, moreover, the closest from start time the measurements we take from the model, the more accurate and reliable the findings will be [1].

Since predictions become more accurate the more recent are the updates, it is crucial for space agencies to observe and track orbiting object as much as possible. Unfortunately, there are simply too few telescopes to watch everything in the sky at all times, here is where propagators models come to place. Space propagation models use updated information of satellites and debris in order to study their path in the near future. Joint use of observation and propagators then allows the space community to efficiently register all objects orbiting earth, while providing means for detection of contacts or even the increasingly frequent collisions [2].

Long time ago, Isaac Newton analytically demonstrated by solving the *two-body-problem* that the path of an orbiting object around a central body describes

an ellipse as Kepler precisely measured. In case of a satellite, this central body is the Earth. However, due to gravitational forces caused by the Sun and other celestial bodies, plus the oblateness of the planet caused by rotation, the satellite will deviate from traditional Kepler orbits observed for planets. This effect becomes more dramatic as altitudes get lower which is the case for most of human made orbiting objects. Furthermore, due to the large cross-section to mass ratio (compared to planets), satellites are significantly affected by solar radiation pressure, specially those with large area solar arrays. Moreover, air drag must also be estimated for low orbiting satellite interacting with residual atmosphere below 800Km.

Low fidelity propagators approximate this effects while disregarding others. High fidelity propagators solve Newton's equations by numerical methods, while low fidelity tend to be rather analytic by implementing formulas. The former are generally recommended for design and analysis phases where quick response and results are valuable; while the latter are more appropriate when accuracy is a premium. When this numerical or analytical models are used to propagate several objects in space and time, the computation power requirements can quickly become prohibitive. This opens the way to investigate computation alternatives to evacuate this processing needs, probably in a parallel fashion.

Modern processor architectures have embraced parallelism as an important pathway to increased performance. Central Processing Units (CPUs) now improve performance by adding multiple cores. Graphics Processing Units (GPUs) have also evolved from fixed function rendering devices into high performance programmable multi-core processors that performs way beyond their intended graphics capabilities. This architectures are what the heterogeneous system concept describes: powerful, but very different processing elements coexisting in the same platform. However, the different nature of the formers make difficult, if not impossible, for developers to take full advantage of the complete processing power available in many modern computer systems. This is the reason that moved Intel, AMD, Apple, IBM, Nvidia, among others, to join as part of Khronos Group, an independent standards consortium, with the aim of creating OpenCL, the Open Computing Language. OpenCL is an open industry standard for efficiently programming heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform, ranging from hand-held devices, PCs, mainframe servers, and even FPGAs [3].

The flexibility of parallel programing OpenCL offers motivated our work: an OpenCL based SGP4 algorithm implementation to cope with the tracking of large number of orbiting objects. Such an implementation can easily be ported to a home ranged computer system, GPUs, a cluster of servers, or even *field programmable gate array* (FPGA) based hardware acceleration. Implementing propagation algorithms received scarce attention from the community, specially, for heterogeneous systems. [4] surveyed different algorithm under specific cluster configurations, and Satellite Tool Kit (STK) [5], probably the most popular commercial tool for satellite propagation and analysis, supports parallel processing, but strictly limited to local CPU or Cluster configuration as well. We review

SGP4 model and OpenCL architecture in Section 2 and 3 respectively in order to introduce the implementation details in Section 4 while finally revising the performance results in Section 5 and concluding in 6.

2 SGP4 Propagator Model

Simplified General Perturbations models such as SGP4 and SDP4 provides orbital state vectors for satellite and space debris referenced to Earth Center Inertial (ECI) coordinate system based on classic orbital elements [6]. SGP4 was developed by Ken Cranford in 1969 and improved in 1979 [7, 8] and includes analytical gravitational and atmospheric models for near-earth (period less than 225 minutes) orbiting elements providing accurate results, without significantly increasing computer time requirements. Later on 1977, deep space models were developed, where solar/lunar perturbations have a larger effect than atmospheric drag, that came to be known as SDP4 [10]. Current code libraries have merged SGP4 and SDP4 algorithms into a single codebase handling the range of orbital periods which are usually referred to generically as SGP4. This code and algorithms were documented and made available in 1980 to the public in Space Track Report #3. David Vallado working through the Center for Space, released an AIAA paper in 2006 [9], which attempted to revise and reconcile the many codes into one standardized code, now available through Celestrak [11] and the reference for the OpenCL implementation proposed in this work. [9] states that SGP4 model has an error of 1 km at epoch and grows at 1–3 km per day.

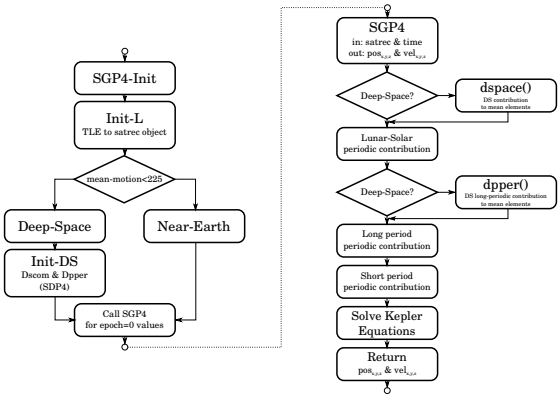

Fig. 1. SGP4 Algorithm

The model takes a two line element set (TLE) character string data as input to initialize SGP4-4 variables; also, gravitational constants are determined at this step. The TLE format, specified in [9] and illustrated in Figure 2, was chosen since it allows for straightforward data import from publicly available and

updated orbiting elements database (over 12,000 space objects are monitored and available at [11]). Afterwards, orbits are initialized at specified epoch, leaving SGP4 main propagator function ready for use.

The SGP4-4 routine takes the initialized structure and time from epoch to propagate to, in order to calculate the position and velocity vector of the spacecraft. Figure 1 depicts the behavior og the algorithm. The precise equations and procedures involved in each step of this calculation are out of scope of this paper and can be found on [12].

Our OpenCL host code implements the main SGP4 routine and uses the aforementioned initialized structures. There is no limit on the number of bodies to propagate in the proposed OpenCL program. It should be noticed that since this parameter set are valid based on a reference time (epoch), the proposed program initially synchronizes (propagates back or forward) all objects to the time specified in the first TLE in the file. The most relevant parameters are detailed below and depicted in Figure 3. We provide further implementation details in Section 4.

- *BStar:* Drag coefficient representing how susceptible an object is to drag.
- *Right Ascension of the Ascending Node:* RAAN is the angle from Aries constellation as a reference longitude to the direction of the ascending node (point where the body crosses the equator from south to north) measured in a reference plane (equatorial).
- *Eccentricity:* Unit-less value with an assumed leading decimal point that determines the amount by which the orbit derivates from a perfect circle (0 is perfectly circular and 1 is parabolic).
- *Argument of Perigee:* The angle between the orbit perigee (closest point to the center) and the ascending node
- *Mean Anomaly:* Relates position and time of a body in a Kepler orbit, goes from 0 to 2π, and it is not an angle, but proportional to the area swept from the focus to body line from perigee which is equal in equal time intervals.
- *Mean motion:* Measured in revolutions per day, if eccentricity is different than 0 it is rather an average value than a instantaneous angular velocity.
- *Revolutions at Epochs:* The number of orbits the body has made since its launch.

Card #	Satellite Number	Class	International Designator			Yr	Epoch Day of Year (plus fraction)	Mean motion derivative (rev/day /2)	Mean motion second derivative (rev/day2 /6)	Bstar (/ER)	Eph	Elem num	Chk Sum
			Year	Lch#	Piece			S	S.	S. S E			
1	1 6 6 0 9	U	8 6 0 1 7	A		9 3 3 5 2 . 5 3 5 0 2 9 3 4		. 0 0 0 0 7 8 8 9	0 0 0 0 0 - 0	1 0 5 2 9 - 3	0	3 4	2

		Inclination (deg)	Right Ascension of the Node (deg)	Eccentricity	Arg of Perigee (deg)	Mean Anomaly (deg)	Mean Motion (rev/day)	Epoch Rev	Chk
2	1 6 6 0 9	5 1 . 6 1 9 0	1 3 . 3 3 4 0	0 0 0 5 7 7 0	1 0 2 . 5 6 8 0	2 5 7 . 5 9 5 0	1 5 . 5 9 1 1 4 0 7	0 4 4 7 8	6 9

Fig. 2. TLE file format

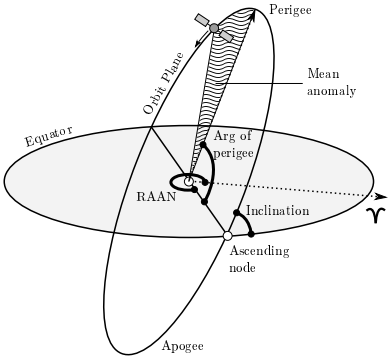

Fig. 3. Orbital Parameters

3 OpenCL Framework

High performance parallel computing was something exclusive for expensive specialized hardware some years ago. But now we can find powerful parallel processors in many home graphics card whose interface has been recently opened by many manufacturers for general purpose computing. OpenCL [13], created by the world most important processors manufacturers, went a little further, aiming for a platform and vendor-independent parallel language. This new processing paradigm is challenging and critical for future computation demanding applications.

Closely resembling the proprietary CUDA language for GPUs from NVIDIA [14], OpenCL is an open industry standard for programming heterogeneous collection of CPUs, GPUs and other discrete computing devices organized into a single platform, ranging from hand-held devices, PCs, up to mainframe servers. OpenCL allows developers to assign specifics tasks of a given program to the most suitable and convenient processing unit, while keeping the coding language unchanged. On the other hand OpenCL was specifically designed to fully explode parallelism within a same device -such as a GPU- and within a same platform, allowing the execution of tasks in many devices at the same time. OpenCL is more than a language; it is a cross-platform, cross-vendor framework for parallel programming.

This way, each OpenCL compatible device accounts with its own specific OpenCL implementation that interprets at run-time the OpenCL language. Then, it is up to each manufacturer to create the device drivers aiming take the best computing power of their product. OpenCL drivers are, at the time of this writing, being published for IBM Cell processors (Sony PS3 current processor), ATI GPUs, Nvidia GPUs, AMD CPUs, Intel CPUs, a wide range of DSPs, and many others. The main target of OpenCL is to write portable yet efficient parallel code, transcending the everlasting trade-off between these two characteristics.

In our case, the CPU acts as a manager for the different SGP4 algorithms being executed in a high performance device such as a GPU. We describe how we mapped our application requirements to OpenCL architecture in the following Sections 3.1 and 3.2.

3.1 Execution Model

Execution model comprises two components: *Kernels* (similar to a C function, it runs on one or more OpenCL devices) and *host* program (executes on the host system, define and interacts with the different devices and is the responsible for managing the kernels executions).

The host program sets up and manages the execution of kernels on OpenCL devices and memory transfers through the use of *contexts* and is written in any code the host system supports: C and C++ are widely used for directly accessing the OpenCL API in different OS. In the other hand, wrappers like: Cloo [15], OpenCLTemplate [16], OpenCL.NET [17], and OpenTK [18], among others are available. In the proposed work, we implemented the host code in C++ for a direct API access.

Kernels, on the other side, are small portions of code, similar to a C function, written in OpenCL C language specification. Kernels code are usually declared in the host code as a constant literal or string, or read from a file (.cl is a common extension). Then, the host program requests the OpenCL API to compile the code at runtime using the API calls described in the OpenCL. The OpenCL runtime Implementation (provided by the manufacturer of each device) will compile and build the code for a specific device requested by the host. The compilation is device dependent, but since it is done at run time, the code becomes fully portable while maximizing performance.

3.2 Memory Model

As common memory between the host and the kernel is unavailable, memory management must be explicit to allow data sharing between the host and the device. Memory types are illustrated in Figure 4 and described below.

- *Host memory:* is the memory available for the host program.
- *Global memory:* is a memory region in which all work-items and work-group can have read and write access both on the host and the compute device. This memory needs to be allocated during run-time. For GPU compute, this is typically the frame buffer memory.
- *Local memory:* Used for data sharing among all work-items in a same work-group. In GPU computing this is usually the local data store for one of the compute units on the GPU.
- *Private memory:* is a region that is only accessible to only one work-item.

The host application has access only to the device global memory: accessing local memory from host memory is no allowed by the OpenCL specification.

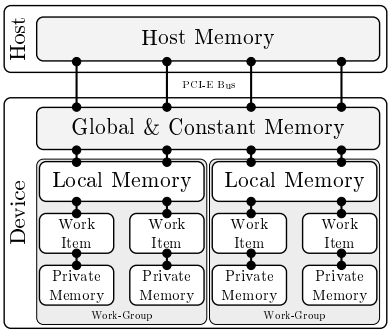

Fig. 4. OpenCL Memory Model

If memory transfers need to be done directly to Local Memory for instance, OpenCL driver will first transfer it to Global and then to Local; the same remains true for the inverse path. Data transfers between the host memory and the OpenCL device generally goes over a PCIe channel or other bus structure that despite providing high data-rates, they fail to achieve the bandwidths available within a device memory and processor. This single fact is critical for proper performance optimization: minimizing host to and from device data transfers during program execution is mandatory to avoid bottlenecks. In general, it is considered a very good practice to transfer as much data as possible to the device memory (and keep it there) before kernel execution. Memory capacity is usually not a problem for an average application in a typical home system since many modern GPU provides, at least, 1 Gbyte of RAM memory. In Section 4 we describe how we incorporate this design principles in implementing SGP4 in OpenCL.

4 Implementation Details

Our current approach was to, in the one hand, generate a C++ based host code capable of initialize OpenCL environment, initialize the SGP4 variables for each object and passing them to the OpenCL API, control the execution of each algorithm (kernels) in the parallel capable device, and finally cleaning up memories and registries. And, in the other hand, a OpenCL kernel code with the SGP4 algorithm itself, that works on the provided orbital elements to provide a ephemeris position and velocity of the object in the time requested. The work-flow is illustrated in Figure 5.

In particular, initializing the environment involves querying the local machine about the OpenCL capable devices available to work with and then decide which of them best suit the SGP4 kernel (this is generally a GPU if any, or a multi-core CPU). Initializing SGP4 variables implies to parse the TLE list provided as input to generate data objects that SGP4 algorithm is capable of interpret. There will be as many orbital objects as TLE elements in the file. This parameters are

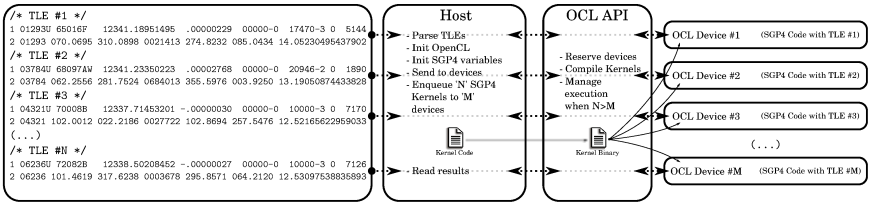

Fig. 5. Application work-flow

sent to the parallel device via OpenCL commands on the initial phase and kept there as long the simulation runs in order to met the design principle described in Section 3.2. The final cleanup frees the occupied memory in the reserved OpenCL device. The SGP4 kernel execution is then similar to a function call, with the difference that this is processed in a more capable device in the platform.

4.1 Floating Point Variables Considerations

Unfortunately, support of double precision floating point variables is not mandatory for OpenCL devices as per Specification [13]. This is reasonable for a compatibility focused framework but might result in accuracy loss for double precision based algorithms such as SGP4. In particular, it is well known that GPUs are typically designed for float variables due to geometry and 3D visualization calculations requirements, which runs short for the precision on angles and trigonometric calculation performed in space mechanics. However, the specification allows this kind of support as a potential extension of a device capability a given host can query and plan upon. We assessed this issue by providing two kernel implementations with a float and double based precision, while providing the host with the intelligence to discover which device supports such variables in order to assign the proper OpenCL code.

We further study this issue and performed basic accuracy measurements to compare the precision loss as the model propagates with the different kind of variables in OpenCL kernels. This analysis came handy since it allowed us to initially contrast both kernel implementations against the original C++ code provided by Celestrak. Figure 6 depicts this analysis evidencing no significant averaged position error -in contrast with the 1 to 3 Km per day stated by [9]- respect the original code within a year (8760hs) propagation time for 500 orbiting elements.

4.2 Time Step Advance

Whichever is the ultimate goal of the OpenCL based SGP4 algorithm, one may need to time advance be fixed or dynamic. The former is the case where the user just need the final position and velocity, while the latter tend to be the scenario for a rather continuous time simulation where dynamic adjustments in time-step is required during the propagation lifetime. Despite this difference seems to be

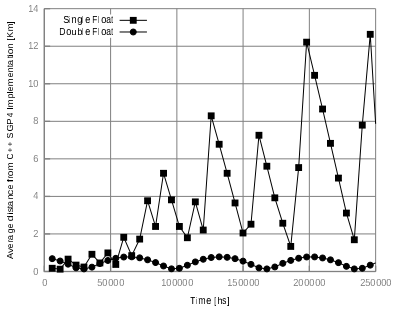

Fig. 6. Average OpenCL-SGP4 Position Error due to Floating Point Type

trivial from an implementation perspective, it is not if we consider the "minimize host to device" data transfer explained in Section 3.2.

As the host code initializes SGP4 variables, it stores them in the device global memory. As the program execution continues, they remain in such space while being updated as the Kernels executes and works over them. The only variables that might (not necessarily) be transfered back to the host memory space are the resulting position and velocity vectors. Likewise, the time advance variable must be informed to the algorithm before execution either through the host program or derived by the Kernel itself. Performance difference is noticeable among this two alternatives since the former involves a data transfer to the device and other does not. Figure 7 shows a measured improvement of 15,3 % in average for a PCIe based GPU. Non noticeable difference is detected for CPU based OpenCL implementations since they does not imply data transfer due to the fact the host and kernel codes are executed on the same hardware.

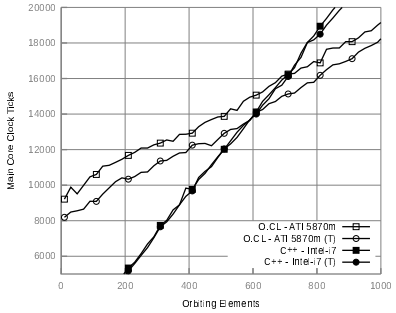

Fig. 7. Performance improvement when avoiding Time Step transfer

4.3 Performance Measurement

In order to evaluate the implementation we need to gauge execution time. Time measurement is a complex, hardware dependent computing topic -specially if milliseconds accuracy is required-. This can turn even worse as OpenCL by nature executes peaces of code in different devices. To assess this issue we included time.h library in the host program and coded OpenCL API functions in blocking mode to force the host to "wait" the return call (i.e. the complete execution of the command). However, it is known that no ANSI C function provides better than 1 second time resolution, and time.h is not the exception. To overcome this lack of accuracy, measurements were performed on long -over 10 minutes long- runs with several kernel execution calls to rely on averaged results. Furthermore, since clock per second constant are hardware dependent, we normalized clocks measurement to single execution loop clocks ticks for comparison purposes.

5 Performance Evaluation

5.1 Benchmarking Hardware

Table 1. OpenCL Benchmarking Hardware

Vendor	Device	Type	Platform Version	Platform Vendor	Comp. Units	Max Dim.	Max WI in a Dim.	Max WG Size	Max Clk Freq	Addr. Bits	Global Mem Size	Local Mem Size	Double F. Point
Intel(R)	Core(TM) i7 CPU Q 720 @ 1.60GHz	CPU	OpenCL 1.1 AMD-APP-SDK-v2.5	Advanced Micro Devices, Inc.	8	3	1024	1024	1596MHz	32	2GB	32KB	Yes
Intel(R)	Core(TM) i3-2330M CPU @ 2.20GHz	CPU	OpenCL 1.1 LINUX	Intel(R) Corporation	4	3	1024	1024	2200MHz	64	4GB	32KB	Yes
AMD(R)	ATI HD 5870 Mobility	GPU	OpenCL 1.1 AMD-APP-SDK-v2.5	Advanced Micro Devices, Inc.	10	3	256	256	700MHz	32	1GB	32KB	No
NVIDIA(R)	GeForce 9500 GT	GPU	OpenCL 1.1 CUDA 4.1.1	NVIDIA Corporation	4	3	512	512	1350MHZ	32	512MB	16KB	No

As sated previously, one of the benefits of OpenCL is the hardware abstraction without performance losses. This allows to easily evaluate the algorithm in different devices aside from it processor architecture while keeping the code intact. We set up a test bed of three systems: an Intel-i7 CPU with an ATI 5870m GPU; a Pentium-IV with a GeForce 9500 GT; and a Intel-i3 laptop; summarizing 5 processing element: two massively parallel OpenCL capable GPUs, two parallel OpenCL capable CPUs, and a non parallel non-OpenCL capable CPU. The hardware selection for the evaluation test bed is diverse to emphasize the heterogeneous system concept OpenCL is based upon. It is interesting to notice how different devices coexists within a same platform although their nature and vendors differs markedly. Table 1 summarizes the test-bed characteristic.

5.2 Scenario

In order to exhibit parallel programming benefits, we measured propagation performance (calculation time) as the number of orbiting elements increases.

Being TLE publicly available, a 1600 real satellite database was created, ranging from low, medium, and geostationary orbit objects in pursuance of studying the behavior of the algorithm over a variety of input parameters. The host code randomly selects a sub set to work with, initializes SGP4 variables, and store them in device memory space.

All the scenarios propagates space nodes for one year time with an hour time step, generating 8760 kernel execution for each orbiting element. For each completed algorithm run, the kernels results -object position and velocity- are queued to be transfered to host program in furtherance of storing them for analysis. A total of 80 scenarios were generated with and increasing step of 20 number of satellites, ranging from 20 up to 1600 total elements. The execution time of 20 propagation is taken as the relative comparison unit we base our measure on with the aim of become independent from device related timings.

5.3 Results

Since Intel and AMD are both providing OpenCL drivers for CPUs parallel devices as well, we were able to test the OpenCL based proposed scenarios not only in GPUs but in i3 and i7 processors. The results are depicted in Figure 8 from which two conclusions arises from them. In one hand, comparing processing elements that typically coexist within a single platform (Intel i7 and ATI 5870, or, Intel P-IV and GeForce 6500), the performance improvement we can achieve just by correctly assigning task to the more capable device is measured in 137,74 and 147,99 times respectively for the extreme 1600 orbiting element scenario. In the other hand, the processing time gain we can expect within a single OpenCL capable device (Intel i7 and Intel i3), by implementing parallel capable OpenCL instead of classic and serial C++ (non-parallel optimized compilation) is proven to be 141,30 and 91,63 respectively in the 1600 satellite scenario as well.

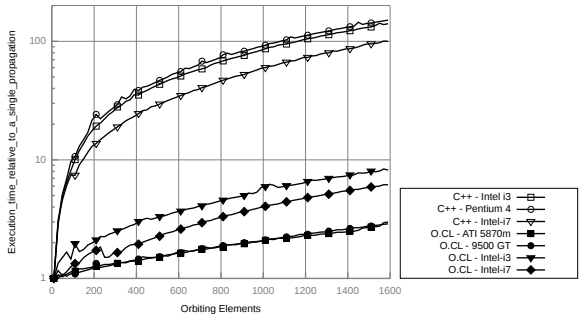

Fig. 8. OpenCL and C++ SGP4 Algorithm Performance Comparison

6 Conclusion

In this work we identified the benefits of augmenting space objects propagation processing performance for the sake of tracking the ever increasing orbiting satellites, spacecrafts or debris. Despite several analysis tools exists, non of them explicitly exploit parallel calculation for propagation algorithm the authors are aware of.

We decided then to implement a popular analytic propagation method known as simplified general perturbations No. 4 or SGP4, in OpenCL, a novel programming framework for heterogeneous processing systems. Although some considerations and simplifications were taken, no significant calculation differences were detected from the original SGP4 algorithm.

Performance results evidenced important gain respect to currently available C++ SGP4 version, particularly in high orbiting elements number scenarios. In these, our implementation outperform the reference one by 147,99 times within a typical platform and 141,30 times within a single processing device.

References

1. Wnuk, E.: Accuracy of Predicted Earth's Artificial Satellite Orbits, Adv. Space Res., 16, (12)lOl-(12)104, 1995.
2. Weeden, B.: Billiards in space. In: The Space Review, February 2009. http://www.thespacereview.com/article/1314/1
3. Fraire, J. A., Ferreyra, A., Marques, C.: OpenCL Overview, Implementation, and Performance Comparison. In: Latin America Transactions, IEEE (Revista IEEE America Latina) , vol.11, no.1, pp.274,280, Feb. 2013
4. Neta, B. et al.: Performance of Analytic Orbit Propagators on a Hypercube and a Workstation Cluster, In: AIAA/AAS Astrodynamics Conference, 1994.
5. AGI - Satellite Toolkit www.agi.com/stk
6. J. Meeus: Astronomical Algorithms, Willmann-Bell, Richmond, Virginia, 1991
7. Lane, M.H. and Cranford, K.H., An Improved Analytical Drag Theory for the Artificial Satellite Problem. In: AIAA Paper Number 69-925, August 1969.
8. Lane, M.H. and Hoots, F.R.: General Perturbations Theories Derived from the 1965 Lane Drag Theory. In: Project Space Track Report No. 2, December 1979, Aerospace Defense Command, Peterson AFB, CO.
9. Vallado, D. A., et al.: Revisiting Spacetrack Report #3. In: AIAA 2006-6753, Center for Space Standards and Innovation, Colorado Springs, Colorado, 80920, 2006.
10. Hujsak, R.S.: A Restricted Four Body Solution for Resonating Satellites with an Oblate Earth, In: AIAA 79-136, June 1979.
11. Celestrak http://www.celestrak.com/
12. Hoots, F. R., Roehrich, R. L.: Spacetrack Report #3, Models for Propagation of the NORAD Element Sets. In: AIAA, U.S. Air Force, CO., 1980.
13. OpenCL Specification v1.1 – Khronos Group (Revision 36, Sep 30, 2010). In: http://www.khronos.org/registry/cl/.
14. T. R. Halfhill,: Parallel Processing with CUDA. In: Microprocessor Report, 2008.
15. Cloo, Dec 2012. http://cloo.sourceforge.net/
16. OpenCL Template, Dec 2012, CMSoft. http://www.cmsoft.com.br/
17. OpenCL.NET, Nov 2011. http://openclnet.codeplex.com/
18. OpenTK http://www.opentk.com/