High Performance Orbital Propagation Using a Generic Software Architecture

M. Möckel*
SERC Limited, Mount Stromlo Observatory, Cotter Road, Weston Creek, ACT 2611, Australia
J. Bennett
SERC Limited, Mount Stromlo Observatory, Cotter Road, Weston Creek, ACT 2611, Australia
E. Stoll
Institute of Space Systems, Technische Universität Braunschweig, Germany
K. Zhang
SPACE Research Centre, School of Mathematical and Geospatial Sciences, RMIT University, GPO Box 2476, Melbourne, VIC 3001, Australia

ABSTRACT

Orbital propagation is a key element in many fields of space research. Over the decades, scientists have developed numerous orbit propagation algorithms, often tailored to specific use cases that vary in available input data, desired output as well as demands of execution speed and accuracy. Conjunction assessments, for example, require highly accurate propagations of a relatively small number of objects while statistical analyses of the (untracked) space debris population need a propagator that can process large numbers of objects in a short time with only medium accuracy. Especially in the latter case, a significant increase of computation speed can be achieved by using graphics processors, devices that are designed to process hundreds or thousands of calculations in parallel. In this paper, an analytical propagator is introduced that uses graphics processing to reduce the run time for propagating a large space debris population from several hours to minutes with only a minor loss of accuracy. A second performance analysis is conducted on a parallelised version of the popular SGP4 algorithm. It is discussed how these modifications can be applied to more accurate numerical propagators. Both programs are implemented using a generic, plugin-based software architecture designed for straightforward integration of propagators into other software tools. It is shown how this architecture can be used to easily integrate, compare and combine different orbital propagators, both CPU- and GPU-based.

1. INTRODUCTION

Orbital propagation, i.e. the calculation of a satellite's position and velocity based on Kepler's equations as well as several perturbation forces, is a common problem in many space science applications. Running such algorithms on a computer always requires a trade-off between speed and accuracy; the higher the accuracy of a perturbation force model, the higher the computation time required. This issue is becoming especially significant when such operations have to be carried out on a large number of satellites. Conjunction assessments and long-term statistical analysis of the object population around Earth are two examples where this is required. The fact that the satellites (or satellite pairs, in the case of conjunction assessment) can be treated independently from each other allows the propagations to be executed in parallel. This enables them to be run on graphics processing units (GPUs) which are optimized for massive parallelism. It was shown in [1] that the computational time of analytical orbital propagators can be greatly reduced by running them on a graphics processor. Since this involves different programming techniques, a software interface for orbital propagators, both CPU- and GPU-based, was introduced that allows for their easy integration into other applications. This paper summarizes the work conducted in [1] and expands upon it with a focus on a greater variety of propagators as well as the additional use case of conjunction assessment.

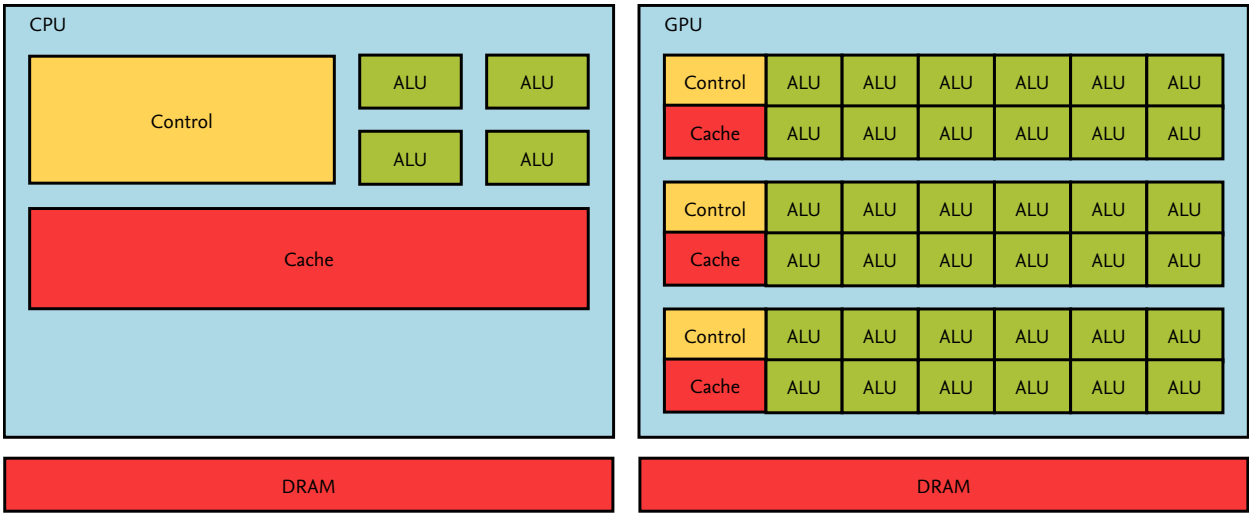

Fig. 1. Illustration of the architectural differences between a CPU and a GPU.

2. GPU COMPUTING

The hardware of graphics processors differs form regular CPUs to enable massively parallel executions. Fig. 1 illustrates the differences between CPU and GPU hardware. The CPU's higher dedication of transistor count towards cache memory and control unit results in a higher versatility, but the number of arithmetic-logic units (ALUs) is much lower in comparison. This means that a CPU is optimized to run more complex algorithms, but only a few threads at a time. The GPU is designed to run a specific class of algorithms, i.e. 3D vector arithmetics and others that are used in 3D graphics applications, but at a much higher concurrency. Instead of four to eight "cores", GPUs are able to run several hundreds to thousands of threads in parallel. Many computer graphics algorithms follow the *Single Instruction, Multiple Data (SIMD)* principle which states that the same operations are executed on a large data set. For example, when an image is generated from a scene of three-dimensional geometry, the color of each pixel of that image is determined by the same algorithm; the fact that each pixel receives a different color is a result of the individual input data each of them receives. Because all pixels can be treated independently of each other, these operations can be executed in parallel. Many applications in space research follow the same principle: Propagating a population of satellites requires the same algorithm to be run independently on all objects. In conjunction assessment, collision probabilities are calculated individually for a large number of object pairs. Problems like these are very likely to profit from execution on GPUs. Collision detection is a common problem in computer graphics and video game applications, and efficient GPU-based algorithms have been developed in that area (e.g. [2]). However, one drawback of using graphics processors for scientific computing, is accuracy. Since single-precision floating point accuracy is usually sufficient for computer graphics the hardware is optimized this way. While all current platforms also support double precision, it usually comes at a significant loss of performance.

In order to program a graphics processor for general purpose applications, several APIs exist, the most common being the *Compute Unified Device Architecture (CUDA)* and the *Open Compute Library (OpenCL)*. Both work in a similar manner: A GPU program consists of an application written in a regular programming language such as C++ that is run on the CPU, or *host*. Embedded into it is code written in a C-like language with some keywords added to control parallelism and memory management as well as advanced vector data types. This code is uploaded to the GPU (called the *device*) at run time. An algorithm run on the device is called a *kernel*. Input and output data for the kernels is arranged into arrays with up to three dimensions which are uploaded into GPU memory. Multiple instances of a kernel (called *threads* in CUDA or *work-items* in OpenCL) are executed in parallel. Each thread/work-item is assigned a unique ID that can be used as an index into the input and output arrays. Threads are grouped into one- to three-dimensional batches — usually matching the dimension of the data arrays — called *blocks* in CUDA and *work-groups* in OpenCL, which are executed concurrently. The example given in listing 1 shows a kernel that adds two values, a_x

*Email: marekmoeckel@serc.org.au

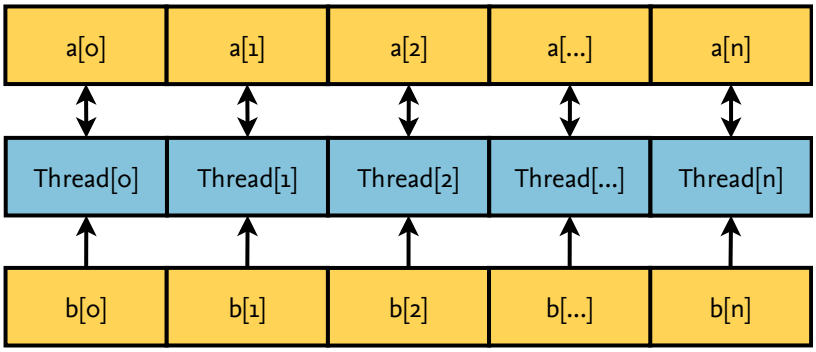

Fig. 2. Illustration of the thread layout for the exemplary kernel.

and b_x, and stores the result in a_x. One thread of this kernel is run for each element of the input arrays a and b, as shown in Fig. 2. The kernel ID, read from *threadIdx.x*, is used as an index into the arrays. The block in this case is one-dimensional with a block size of (n, 1, 1). Parallelism is limited not only by the number of available "cores" but also by memory constraints. Since all concurrent threads have to share the available registers more complex algorithms might not be able to fully utilize the available hardware.

Listing 1. A simple CUDA kernel for adding two integer arrays.

```
// A simple CUDA kernel that adds the values of the two integer arrays a and
// b and stores the result in array a. Every thread of this kernel works on
// one element of each array corresponding to its own index.
__global__ void add(int *a, int *b)
{
  a[threadIdx.x] += b[threadIdx.x];
}
```

CUDA is being developed by Nvidia Corporation and works only on hardware manufactured by that company. OpenCL does not only support graphics processors of multiple vendors but other parallel hardware like multicore CPUs or co-processors such as the Xeon Phi. Contrary to CUDA which builds the GPU binary during compilation, OpenCL kernels are compiled at run time for the platform that was chosen for executing them. Compared to CUDA, this results in a slightly longer run time of the serial component of the program as well as more complex code to account for the different hardware architectures. These disadvantages are made up for by the ability to run OpenCL code on a wide variety of platforms without further modification.

3. SOFTWARE ARCHITECTURE

One of the main practical issues of GPU computing in space research is maintainability. Even though the languages used to program graphics processors is strongly based on C, the underlying hardware is very different from regular CPUs. In order to get the best performance out of a GPU, a good understanding of its design is essential. Additional code is required to manage data transfers from and to GPU memory, kernel execution, and hardware parameters. Orbital propagators are an important everyday tool in many fields of space research; it is therefore desirable to design GPU-based propagators in a way that they can be used, written, and integrated into other software tools by anyone with regular programming skills.

In order to make this possible, the *Orbital Propagation Interface (OPI)* was designed. OPI is a software library that acts as a middleware between orbital propagators and applications that use them, allowing propagators to be designed as independent, interchangable plugins. Host applications and plugins can be written in different languages; OPI currently supports C, C++, Fortran, Python (via C interface) and CUDA for GPU-based propagators. It provides base classes and data types that facilitate the design and implementation of the propagation algorithm. GPU computing

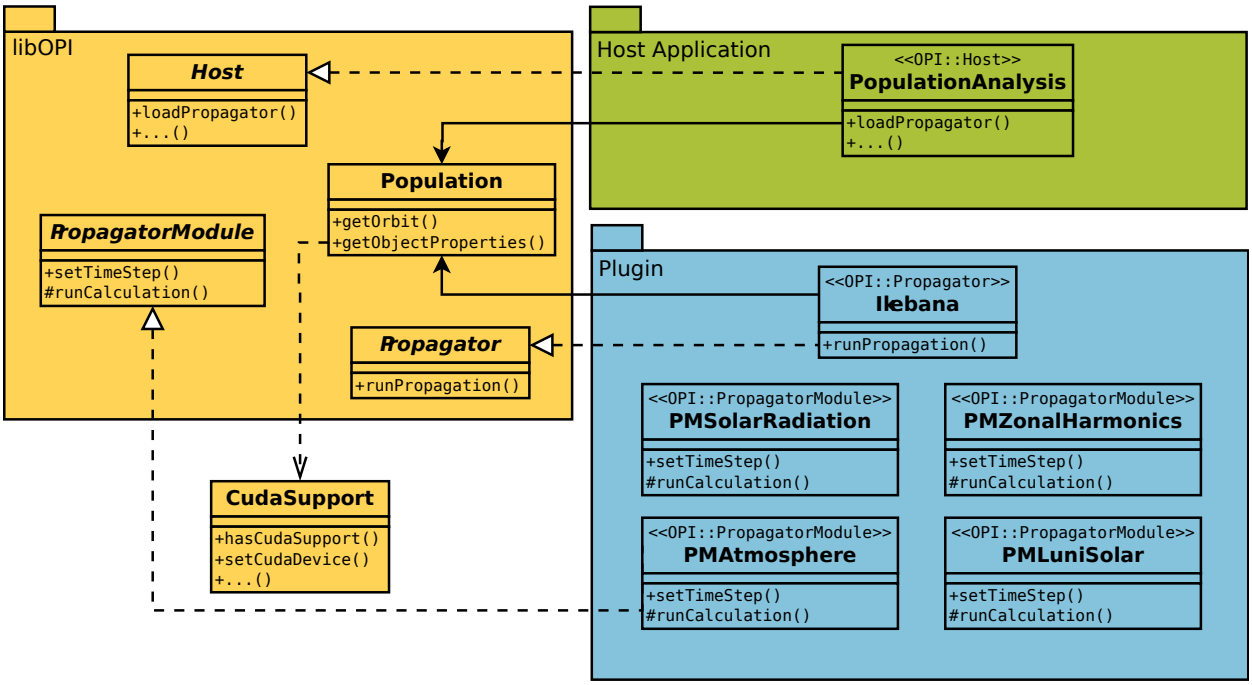

Fig. 3. UML diagram of the Orbital Propagation Interface.

support is integrated in a way that makes it easy to use when designing a propagator, and completely agnostic to the host application. The OPI design is the result of several years of working with different orbital propagators. The interface between host and plugin is based on the smallest common denominator of those propagators' input and output variables, as well as the recommended standard for orbital data messages [3]. It was first implemented as part of a master's thesis [4] and published under an open source license, along with documentation and example code [5]. Fig. 3 shows a UML diagram of OPI. The library provides four main classes named *OPI::Host*, *OPI::Propagator*, *OPI::Population* and *OPI::PropagatorModule*, respectively. The first two provide the main interfaces: The host application implements *OPI::Host* to access functions that search for available plugins, select and load a propagator; the plugin implements the *runPropagation()* function of *OPI::Propagator* in order to make it accessible to the host. *OPI::Population* serves as a data exchange class: It contains a list of satellites represented by their orbits as well as properties such as mass, size, and reflectivity (refer to table 1 for a full list of attributes). The host application creates an instance of *OPI::Population* and fills it with data about the objects that should be propagated. Upon calling the plugin, the population object is read by the propagator and updated with the new state. All communication between CPU and GPU memory is built into OPI's population data types. When running a propagator that uses the GPU for its computations, orbital elements defined by the host in CPU memory are transferred automatically to the GPU once the propagation process is started. Likewise, the results are downloaded into CPU memory once the host requests them. The downloads do not happen automatically after every time step in order to avoid expensive and unnecessary data transfers. The *CudaSupport* class that handles the memory operations is structurally separated from the rest of the interface so it can be left out or replaced with one that supports a different API like OpenCL. *OPI::PropagatorModule* is an optional class that can be used to further break down the structure of a more complex propagator. It provides an interface similar to the *OPI::Propagator* class, but tailored to suit individual perturbation models. Multiple modules can be linked together at compile time to form a complete propagator, or compiled into individual plugins which can be selected individually at run time.

Obviously, a common interface cannot possibly contain all variables that a specific propagator might require to run. The OPI allows the definition of *PropagatorProperties* within propagators and propagator modules; these are variables that can be queried, read and written by the host application. They can be used to provide data such as lookup tables, configuration options and additional results. They were used, for example, in the OPI port of the SGP4 propagator: It is designed specifically to work with the TLE data format, so input cannot be provided via the regular *OPI::Population*

Value	Description	Valid Range
Orbit.semi_major_axis	Semi major axis [km]	$a > 6400$
Orbit.eccentricity	Eccentricity []	$0 < \epsilon < 1$
Orbit.inclination	Inclination [rad]	$0 < i < 2\pi$
Orbit.raan	Right ascension of the ascending node [rad]	$0 < \Omega < 2\pi$
Orbit.arg_of_perigee	Argument of perigee [rad]	$0 < \omega < 2\pi$
Orbit.mean_anomaly	Mean anomaly [rad]	$0 < M < 2\pi$
Orbit.bol	Beginning of life / launch date [MJD]	0 or > 2400000.5
Orbit.eol	End of life / decay date [MJD]	0 or > 2400000.5
ObjectProperties.id	Numerical identifier	$-2^{31} \leq id < 2^{31}$
ObjectProperties.mass	Mass of satellite [kg]	$m > 0$
ObjectProperties.diameter	Satellite's mean diameter [m]	$d > 0$
ObjectProperties.area_to_mass	Satellite's area to mass ratio [$\frac{m^2}{kg}$]	$\frac{A}{m_{sat}} > 0$
ObjectProperties.drag_coefficient	Satellite's drag coefficient []	$C_D > 0$
ObjectProperties.reflectivity	Satellite's reflectivity coefficient []	$0 \leq C_R \leq 2$

Tab. 1. Overview of OPI's population data types. Valid ranges are given as implementation guidelines and are not enforced by OPI.

data type. Instead, a *PropagatorProperty* is defined containing the name of a file which is opened when the propagator is initialized, and the TLE data is read from it. The state vectors and orbital elements generated by SGP4 are written into an *OPI::Population* and output normally via the interface.

4. PERFORMANCE COMPARISON

To measure the speedup of parallelized propagation in different environments, two propagators have been ported to the GPU and compared against their original counterparts. Run time performance is given as

$$\frac{N_{obj} \cdot N_{st}}{t_{total} \cdot 10^6}$$

where N_{obj} is the number of objects that are propagated, N_{st} is the number of propagation steps and t_{total} is the overall run time of all propagation operations in seconds (no data transfers or other operations are measured). The resulting value expresses performance in *megapropagations per second (MP/s)*. It was chosen because it is easy to calculate and provides a good performance overview in both absolute and relative terms. Since it averages over all objects it can depend largely on the composition of the object population being propagated: Objects that are influenced by atmospheric drag usually need more computation time than others. This must be kept in mind when comparing run times; the OPI interface facilitates automated performance tests by ensuring that all propagator plugins are run on the same input data.

The first algorithm to be tested is FLORA, an analytical propagator developed at the Institute of Space Systems for long-term space debris environment analysis and described in [6]. It calculates atmospheric drag based on the NRLMSISE-00 model [7], solar radiation pressure based on the algorithms published by Cook [8] and Escobal[9] as well as third body and gravitational perturbations as described by Vallado [10]. The GPU port, dubbed *Ikebana*, was written in CUDA and executed on three different graphics processors. Ikebana differs from FLORA in that it makes extensive use of OPI both as a design template and as the primary interface; while FLORA has been compiled as an OPI plugin it does not use any of its data types internally. The other significant difference is that Ikebana uses single-precision floating point variables when possible, whereas FLORA uses double precision throughout. It was shown in [1] that for the purpose of statistical long-term analysis of the space debris population, the propagator's accuracy is still sufficient when only single precision is used (Fig. 4). The test scenario for FLORA/Ikebana is based on a recent snapshot of the publicly available TLE population of around 15,000 objects, multiplied with slight parameter variations to increase the number of objects to over 200,000. They were propagated over a time span of 50 years in steps of one day. The tests were conducted using two CPUs (Intel Core i7-3820 and Intel Core i7-4710HQ) and three GPUs (Nvidia

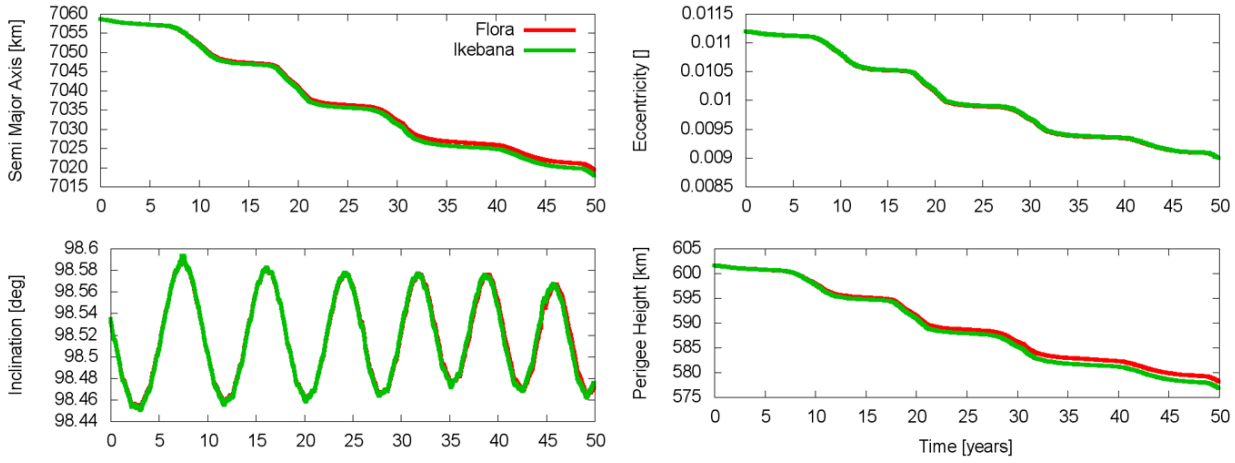

Fig. 4. Accuracy comparison of FLORA and Ikebana.

Tesla K20c, Nvidia GeForce GTX 960 and Nvidia GeForce GTX 860m). As shown in Fig. 5, Ikebana shows a huge performance improvement over FLORA. While the CPUs calculated between 100,000 and 130,000 propagations per second, the GPUs performed at around 3, 5 and 7.5 MP/s, respectively; every new generation of GPUs ran significantly faster than its predecessor. In relative terms, the fastest GPU showed a speedup of almost 60 compared to the fastest CPU, thereby completing an hour's work in just over a minute. With less than 50,000 objects the GPUs were not able to reach their peak performance because the cards' massively parallel architecture could not be fully exploited. Fig. 6 shows a significant loss of performance on the GPUs when double precision is used. Only the Tesla K20c, which is explicitly designed for scientific computing, does not show this behaviour. However, it is still outperformed by the newer (and much cheaper) consumer-grade GTX 960.

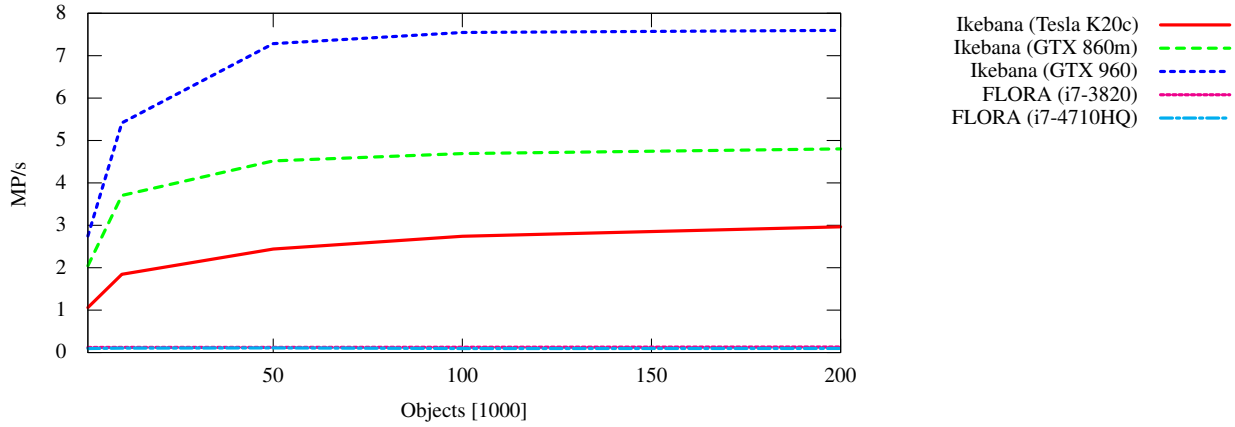

Fig. 5. Performance results for FLORA and Ikebana.

The other propagator that has been tested is the popular SGP4 algorithm. The GPU port is based on the original C++ implementation given in [11] which is also used as the basis for performance comparison. The parallelized version was written in OpenCL without any changes to the original algorithm beyond those that were required for porting. Since this propagator is being used in a conjunction assessment context a loss of accuracy is not acceptable in this case; therefore, the double precision of the original algorithm was retained at the likely cost of GPU performance. OpenCL ports of SGP4 have been done before, particularly in [12] which also confirms the loss of accuracy by using single precision floats. The paper shows performance results for older CPUs and graphics processors which, due to

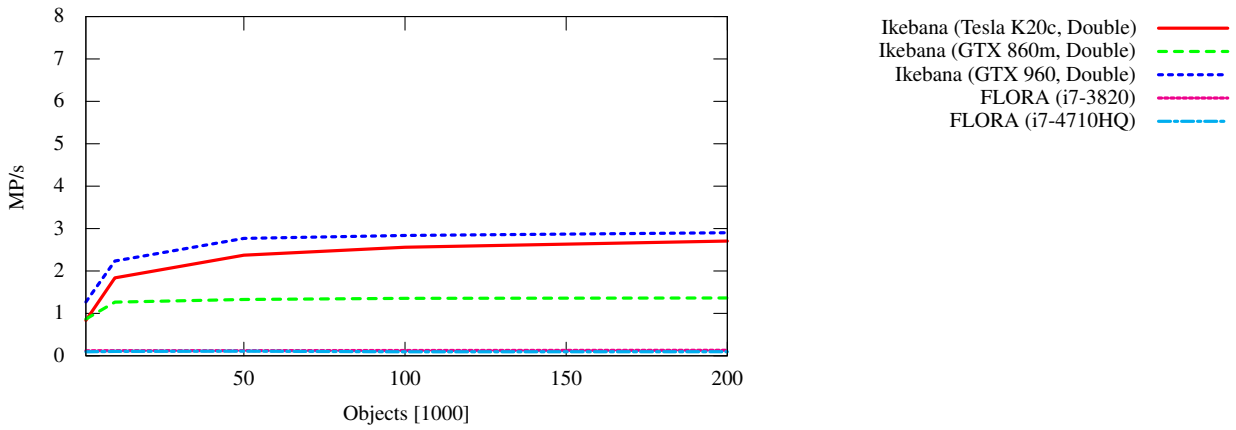

Fig. 6. Performance results for Ikebana with double precision.

fast progress in hardware development, do not necessarily reflect the current situation anymore. Results are given in terms of relative speedup between the tested devices so unfortunately, the results cannot be directly compared to this work without access to the hardware that was used. As for the previous analysis, the test population for SGP4 is based on a recent TLE snapshot of around 15,000 objects, multiplied to create the required number of 200,000 objects. Since SGP4 is not suitable for long-term propagation, a more realistic scenario was chosen that propagates the objects over a time span of seven days with a step size of 200 seconds. The propagation was executed on two different CPUs (Intel Xeon e5-1620 and Intel Core i7-4710HQ) as well as two GPUs (Nvidia GeForce GTX 960 and Nvidia GeForce GTX 860m). Since OpenCL supports multi-core processors, the parallelized propagator was also run on the Xeon processor using all eight cores. Performance results for SGP4 are shown in Fig. 7. Like Ikebana, the propagator benefits greatly from parallelization on the GPU. While both CPUs perform fairly stable at just under 1 MP/s, the GPUs reach around 6 and 11 MP/s, respectively. The object number at which the relative overhead is negligible is around 30,000 for the GPUs. The OpenCL code also works well on the multi-core CPU, reaching around 6.5 MP/s and outperforming the slower GPU. This value amounts to a relative speedup of around 7.2 compared to a single core on the same CPU, very close to the theoretical maximum of 8. Unlike the graphics processors, the CPU does not take a performance hit from running the propagator in double precision; still, the faster GPU almost doubles the speed of the fully utilized CPU. Compared to [12], the relative speedup between the fastest CPU and the fastest GPU differs greatly from the factor of over 140 achieved by the authors. Differences in hardware, OpenCL version, absolute performance, input data, optimization techniques, and use of double instead of single precision are all likely factors contributing to this outcome.

5. OUTLOOK: CONJUNCTION ASSESSMENT AND NUMERICAL PROPAGATION

One of the goals behind this research is to decrease the run time of the collision assessment procedure. Currently, SGP4 is used to propagate a given object catalogue into the future, and a full conjunction analysis is performed in each time step. While SGP4 only makes up for around two per cent of the total run time (as shown in Fig. 8), early tests showed that the whole collision assessment process can be sped up significantly when using the GPU, reducing the total run time of a conjunction analysis from over two hours to just fifteen minutes. Due to the GPU's memory constraints simple prefilters such as a basic perigee-apogee proximity test as described by Hoots et al. [13] achieve a much higher rate of parallelism than more in-depth tests. Fig. 9 shows the performance results of such a filter, achieving a speedup factor of around 30 when approximately 35% of object pairs pass the filter, i.e. result in a possible collision. The pass rate can have a significant impact on the overall performance: A high number of false positives from a simple filter leads to more object pairs that need to go through the subsequent filter chain. It also requires more output data to be written which slows down the process, especially when GPU-to-CPU memory transfer is involved. Finding the right balance between complexity of the filter and pass rate plays a more significant role in the optimization process when developing for the GPU.

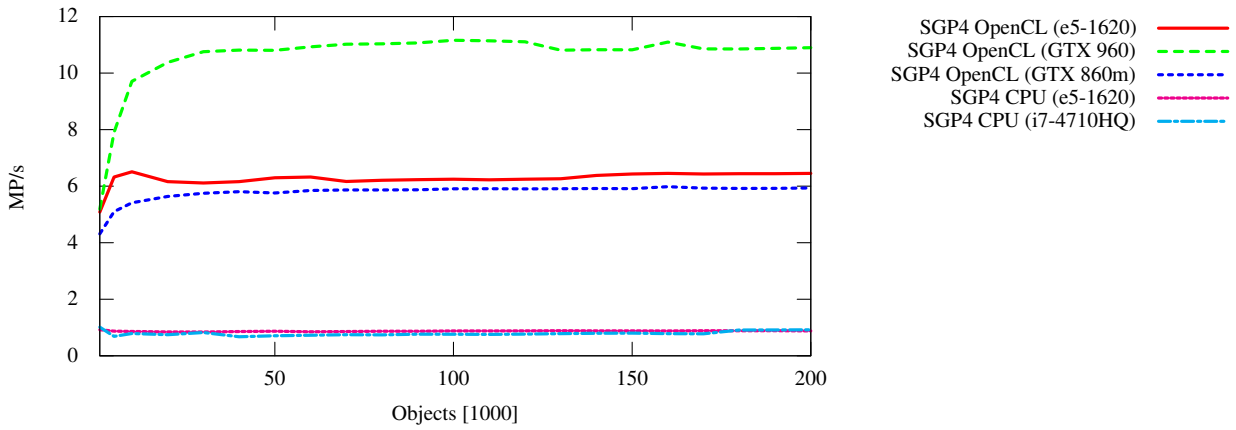

Fig. 7. Performance results for SGP4.

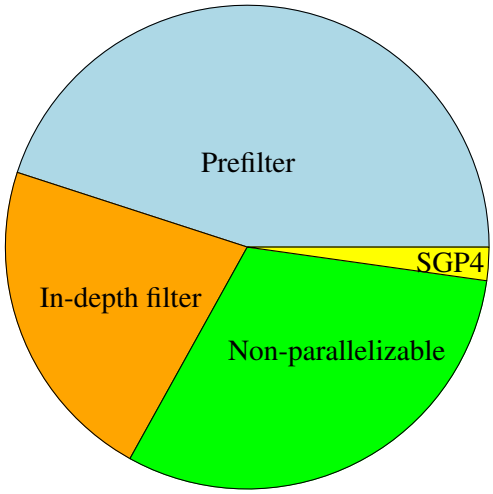

Fig. 8. Run-time distribution of collision assessment tool.

Besides propagators, OPI also supports a second type of plugins for conjunction filters and the calculation of collision probablilities [4]. These tie in with OPI-based hosts and propagators and facilitate the development of modular, GPU-based conjunction assessment tools.

For a more thorough short-term conjunction assessment it is advisable to use more accurate, numerical propagators instead of analytical ones. For their parallelization, the ODE solver is a critical component. For SIMD platforms like graphics processors, fixed-step solvers like Runge-Kutta are generally better suited than variable-step solvers. GPU implementations of the Runge-Kutta method exist and show good improvements over their serial counterparts [14][15]. A case study on a numerical propagator called ZUNIEM [16] shows promising results for parallelization in OpenCL with a variable-step Shampine-Gordon solver. The speedup was more apparent on multi-core CPUs which are not bound by the SIMD paradigm and therefore lend themselves better to variable-step algorithms. It is possible that performance on GPUs might be improved by intelligent sorting, i.e. ensuring that those objects likely to follow the same path through the algorithm are executed in one batch. The work on Ikebana has also shown that careful, selective conversion from double to single precision can provide great performance boosts while still keeping the accuracy loss within acceptable levels. ZUNIEM's internal structure is similar to that of FLORA which makes it suitable for an

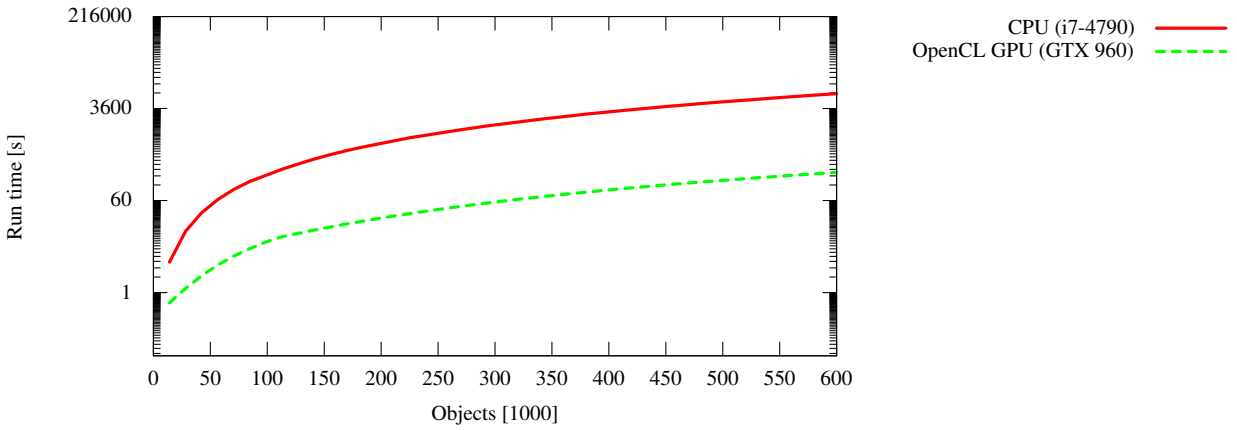

Fig. 9. Performance results for apogee-perigee proximity filter.

OPI-based implementation. For better support of more advanced propagators, OPI needs to be extended to include an interface for covariance matrices, double precision floats, and OpenCL instead of CUDA. The latter has been factored into the OPI's design by concentrating all relevant code into a separate "cuda-support" module that can easily be replaced by one that uses OpenCL. Despite being slightly more complicated to use than CUDA, OpenCL has the great advantage of supporting non-Nvidia platforms, including multi-core CPUs which are more widespread in business environments and have shown good results for orbital propagation.

ACKNOWLEDGEMENTS

The authors would like to acknowledge the support of the Cooperative Research Centre for Space Environment Management (SERC Limited) through the Australian Government's Cooperative Research Centre Programme.

REFERENCES

[1] M. Möckel, *High Performance Propagation of Large Object Populations in Earth Orbits*. PhD thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2015. https://dx.doi.org/10.5281/zenodo.48180.

[2] D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, and M. Mitzenmacher, "Real-Time Parallel Hashing on the GPU," 2006.

[3] CCSDS, "Consultive Committee for Space Data Systems - Orbit Data Messages - Recommended Standard," 2009.

[4] P. Thomsen, "GPU-Basierte Analyse von Kollisionen im Weltraum," Master's thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2013.

[5] P. Thomsen and M. Möckel, "OPI Source Code on GitHub." `https://github.com/ILR/OPI`, 2013. Accessed 2015-08-19.

[6] S. Flegel, "Simulation der zukünftigen Population von Raumfahrtrückständen unter Berücksichtigung von Vermeidungsmaßnahmen," Master's thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2007.

[7] J. Picone, A. Hedin, D. Drob, and A. Aikin, "NRL-MSISE-00 Empirical Model of the Atmosphere: Statistical Comparisons and Scientific Issues," *J. Geophys. Res., doi:10.1029/2002JA009430*, 2002.

[8] G. E. Cook, "Luni-Solar Perturbations of the Orbit of an Earth Satellite," *Geophysical Journal of the Royal Astronomical Society*, 1962.

[9] P. Escobal, *Methods of Orbit Determination*. Krieger Publishing, Inc., 1965.

[10] D. A. Vallado, *Fundamentals of Astrodynamics and Applications*. Microcosm Press, 3rd ed., 2007.

[11] D. Vallado, P. Crawford, R. Hujsak, and T. Kelso, "Revisiting Spacetrack Report #3," *AIAA 2006-6753*, 2006.

[12] J. Fraire, P. Ferreyra, and C. Marques, "OpenCL-Accelerated Simplified General Perturbations 4 Algorithm," *Proceedings of the 14th Argentine Symposium of Technology*, 2013.

[13] F. Hoots, L. Crawford, and R. Roehrich, "An analytic method to determine future close approaches between satellites," 1983.

[14] M. Rodriguez, F. Blesa, and R. Barrio, "OpenCL parallel integration of ordinary differential equations: Applications in computational dynamics," *Computer Physics Communications*, vol. 192, p. 228236, Jul 2015.

[15] W. M. Seen, R. U. Gobithaasan, and K. T. Miura, "GPU acceleration of Runge Kutta-Fehlberg and its comparison with Dormand-Prince method," 2014.

[16] C. Kebschull, "Parallelisierung eines numerischen Propagators mit OpenCL," Master's thesis, Technische Universität Carolo-Wilhelmina zu Braunschweig, 2011.