THE PROCESS OF PARALLELIZING THE CONJUNCTION PREDICTION ALGORITHM OF ESAS SSA CONJUNCTION PREDICTION SERVICE USING GPGPU

M. Fehr, V. Navarro, L. Martin, and E. Fletcher

European Space Astronomy Center, Villanueva de la Cañada, 28691 Madrid, Spain,
Email: {vicente.navarro, luis.martin, emmet.fletcher}@esa.int, marius.b.fehr@gmail.com

ABSTRACT

Space Situational Awareness[8] (SSA) is defined as the comprehensive knowledge, understanding and maintained awareness of the population of space objects, the space environment and existing threats and risks. As ESA's SSA Conjunction Prediction Service (CPS) requires the repetitive application of a processing algorithm against a data set of man-made space objects, it is crucial to exploit the highly parallelizable nature of this problem. Currently the CPS system makes use of OpenMP[7] for parallelization purposes using CPU threads, but only a GPU with its hundreds of cores can fully benefit from such high levels of parallelism. This paper presents the adaptation of several core algorithms[5] of the CPS for general-purpose computing on graphics processing units (GPGPU) using NVIDIAs Compute Unified Device Architecture (CUDA).

Key words: GPGPU; Conjunction Analysis; Space Situational Awareness; Conjunction Prediction Service; Parallel Programming; ESA; ESAC;.

1. INTRODUCTION

As the population of space objects is constantly growing, maintaining Space Situational Awareness (SSA) has become a core competence of many space related companies and organizations. Apart from the space weather and the Near Earth Objects (NEO) segments, the Space Surveillance and Tracking Segment (SST) one of key components in ensuring save access to the space environment and mastering the existing threads/risks. Among other problems SST addresses conjunction prediction analysis, re-entry prediction, fragmentation analysis, catalogue correlation and orbit determination. The required computations are characterized by their demand for a large number of parallel computations in a very short period of time.

To meet the need for independent utilization of and access to space for research or services by an increasing number of users, the European Space Agency is in the process of adapting the European SSA System to offer modern web-based front-end. The system provides the users with timely and quality data, information, services and knowledge regarding the environment, the threats and the sustainable exploitation of the outer space surrounding our planet Earth. Both the ability to provide service to multiple users in parallel and the increasing number objects call for better performance and parallelization of the core algorithms.

This paper presents the adaptation of several core algorithms of ESAs SSA Conjunction Prediction Service (CPS) for general-purpose computing on graphics processing units (GPGPU). The prediction can be split up into smaller mostly independent sub problems, in which exactly one pair of objects is analysed. This is an ideal scenario for parallel programming. Currently the CPS system makes use of OpenMP for parallelization purposes, which uses all available cores in the CPU of the machine. Even though today's multiprocessor CPUs can launch a number of threads simultaneously only a GPU with its hundreds of cores can fully benefit from such high level of parallelism. A single GPU can launch thousands of threads which can lead to a performance gain by several orders of magnitude.

In this paper we describe the development of a prototype that replaces the Conjunction Analysis (CAN) System[4, 6], a part of the Conjunction Prediction System, with GPU enabled code based on NVIDIA's CUDA architecture and C language extension. We will start by describing the choice of GPU technology used for developing and testing the prototype. In order to validate the concept we reused the existing CPS code for data retrieval from database/files, object state vector interpolation and collision risk evaluation. Using cross-language development we seamlessly integrated the prototype (in C CUDA) into the existing system (in Fortran). Hence we were able to compare one to one the output and the performance of both systems (modified and current). Furthermore we describe what methods were used on each parallelizable part of the algorithm and how the strengths and weaknesses of the GPU influenced the design of each part. We conclude with the lessons learned from developing this prototype and present the results obtained by the performance measurements.

Proc. '6th European Conference on Space Debris'
Darmstadt, Germany, 22–25 April 2013 (ESA SP-723, August 2013)

2. CONJUNCTION PREDICTION SERVICE (CPS)

The Conjunction Prediction Service is a core functionality of the Space Surveillance and Tracking mission (SST). The current deployment of the Conjunction Prediction Service has the following structure:

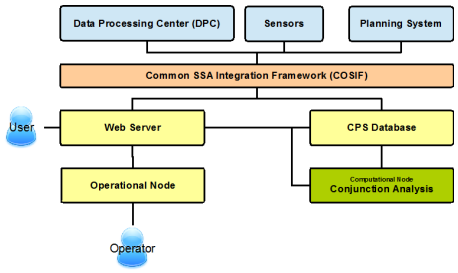

Figure 1. Simplified structure of the CPS system.

The system allows users to trigger their own conjunction prediction computations and access the conjunction database through a web interface. In this paper we focus on the work done by the conjunction computational node, the Conjunction Analysis (CAN) System. This core part of the CPS system, executes the computations necessary to determine potential conjunctions according to the general conditions and settings provided by the users through the web interface and the operators through the operational node. The system should not only be able to withstand a large number of simultaneous requests by users and operators and process them in an appropriate amount of time, but also be ready for the increasing size of catalogues. Therefore the performance of the algorithms involved is essential.

3. NVIDIAS COMPUTE UNIFIED DEVICE ARCHITECTURE (CUDA)

NVIDIA introduced CUDA[3] as software and hardware architecture to allow access to the memory and virtual instruction set of CUDA enabled GPUs. Using this programming model[1, 2], the developer can run highly parallel code on the GPU like on a CPU. The main difference is the large number of parallel stream processors that allow for many concurrent, but slow, threads instead of a small number of fast threads running on powerful multi-purpose cores. As a parallel computing platform, CUDA supplies the developers with programming languages extensions (e.g. CUDA C/C++, CUDA Fortran), directives for optimizing compilers (e.g. OpenACC) and GPU-accelerated libraries (e.g. cuBLAS, cuFFT). We implemented the prototype using the CUDA C/C++ programming language extension, as it is free, very well documented and comes with the official NVIDIA nvcc compiler, whereas some other language extensions (e.g. for Fortran) require expensive 3rd party software.

The basic functionality of CUDA is supported by the G8x series graphics cards onwards, but the range of supported functions on NVIDIA graphics cards varies a lot and can significantly limit the use of CUDA for certain tasks. As the CPS system requires calculations involving state vectors using double precision (location and velocity of objects), we required a NVIDIA graphics card with double precision capabilities and minimal performance penalty. NVIDIA assigns a compute capability ranging from 1.0 to 3.5 to its GPUs to describe the scope of supported functions. Even though double precision support is introduced with compute capability 1.3, the performance penalty for such calculations depends solely on the GPUs architecture (e.g. Fermi, Kepler) and does not necessarily decrease as the compute capability increases. CUDA supports the use of multiple GPUs with one CPU and introduces the term **device** for a single graphics card and **host** for the CPU. The code sent to the devices and executed in parallel is referred to as the **kernel**. GPUs with compute capability higher than 2.0 support the simultaneous execution of multiple kernels. Each GPU, depending on its architecture, contains a different number of streaming multiprocessors (SMs). Each of those SMs contains a specific number of stream processors, each capable of executing one single thread at a time. In order to cope with the varying number of SMs and stream processors among the different GPU architectures, CUDA introduces a general thread model as a layer of abstraction. In order to map threads to the GPUs SMs CUDA organizes threads in an up to three dimensional grid of thread blocks and each block organizes its threads in an up to three dimensional grid. At runtime the GPU automatically distributes the thread blocks among the different SMs to ensure an optimal utilization of all resources. SMs then execute threads in a SIMT (Single Instruction Multiple Threads) fashion in groups of 16 to 32 threads , so called warps. SIMT means that all threads in a warp have to execute the same instruction at the same time. If a warp has to wait for a few cycles (e.g. to load data) it is swapped out and another warp is executed in the meantime. In general this thread model allows the developer to dimension the grid and the thread blocks in the way it best suits his task and create portable code.

4. THE ADAPTATION OF THE CONJUNCTION ANALYSIS (CAN) SYSTEM

The main task of the conjunction analysis is the detection and risk assessment of close encounters between objects in an orbit around earth. The objective is to issue warning bulletin to the appropriate parties to allow for further risk assessment or even orbit corrections if there are manoeuvrable objects involved. Such an analysis could involve only specific subset of the space population during a short time period or the whole object catalogue during an extensive period of time. The latter involves handling large amounts of data and computationally intensive calculations, as the number of pairs to analyse grows quadratically with the catalogue size. The SSA program is currently using the public catalogue provided by the

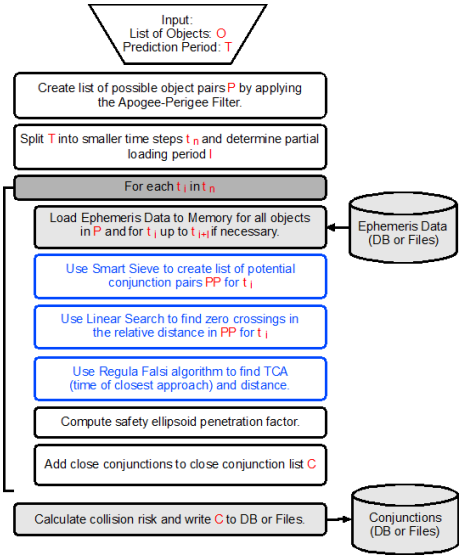

Figure 2. The CAN processing flow

U.S. Air Force JSpOC with 16'000 objects, but there are estimates[9] that more than 600'000 objects bigger than 1cm are in orbit around earth.

Our first objective was to identify all parts of the program that can be parallelized using a GPU and would profit the most. We analysed each subtasks and evaluated their potential. Fig. 2 shows the strucutre of the CAN algorithm. The parts highlighted in **bold** are using OpenMP to parallelize loops on the CPU. Therefore those already parallelized parts are ideal candidates.

- Apogee-Perigee Filter: This is the first filter applied to eliminate all object pairs that have no chance of ever producing a conjunction. After this filter, the prediction period is analysed in smaller time steps, so called epochs, one at the time.
- Loading Ephemeris: Is executed every few epochs to load the required ephemeris data for each object (position an velocity) from database to main memory.
- **Smart Sieve:** This local filter eliminates even more object pairs. The filtering however is only valid for the current epoch.
- **Interpolating For Smart Sieve:** Previous to the Smart Sieve, the state vectors for all objects are updated to the current epoch. The state vector matching the current time (the time of the epoch) is interpolated from the ephemeris data in memory.
- **Linear Search:** The epoch is divided into even smaller time steps to perform a linear search for a sign change of the relative velocity along the relative position vector.
- **Interpolating for Linear Search:** Linear Search requires the exact state vectors at the beginning of each small time step for each object analysed. Hence those vectors are interpolated from the ephemeris data in memory.
- **FindTCA (Time of closest approach):** Given a time step with a sign change from the linear search function, this part applies the Regula Falsi algorithm to find the exact time of the zero crossing and therefore the exact time of closest approach.
- Conjunction Definition: This part collects all the information about each potential conjunctions as a preparation for inserting it into the database.
- Penetration Factor: This part includes the calculation of the safety ellipsoid penetration factor and the final filtering of the close conjunctions. The safety ellipsoid factor gives a measure of how close an object comes to another (e.g. using a safety ellipsoid of 25 x 10 x 10 km).
- Remaining Operations: This part includes all the time spend in between the parts described above.

To further analyse the potential for improvement, we conducted the following performance analysis (fig. 3) of the original program. We ran an all vs. all conjunction prediction involving 10000 objects during 8 days and disabled all OpenMP directives. Thus it appears that the percentage of the overall runtime spent in the parts using OpenMP further supports our initial selections of candidates. The interpolation in the local and in the global filter are basically the same function executed at different steps of the computation and therefore the interpolation in the global filter obviously becomes a candidate too even though it does not contribute greatly to the overall runtime.

4.1. Cross-Language Development

The original conjunction prediction program is mainly written in Fortran 90. As we wanted to use the CUDA C extension, we decided to translate the Fortran module that contains the functionality described above to C. This module, called CANfi_Filters, is part of the main conjunction analysis library. In our prototype it is moved to a separate library. This new library not only contains a C version of the CANfi_Filters module but some C versions/copies of functions from other libraries as well. This was necessary as calling a function in another language is not supported from inside a CUDA kernel (the code executed on the GPU). Fig. 4 shows the changes in the structure of CAN system.

4.2. Non-GPGPU Algorithm Optimizations

In the course of translating the CANfi_Filters module from Fortran to C we discovered two major non-GPU re-

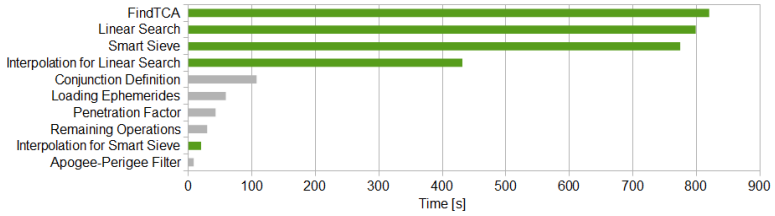

Figure 3. Time spent on each section of the algorithm

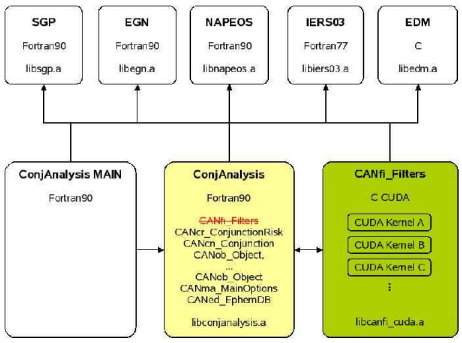

Figure 4. Structure of modified CAN system

lated optimizations in the original program. We decided to implement these changes in our prototype and we will briefly describe them as they had a non-negligible impact on the performance and omitting it would distort the performance measurements.

At the end of each epoch the conjunctions found are added to a temporary list of conjunctions. This list is then analysed and each conjunction is assigned a penetration factor. Only the pairs where the penetration factor is positive, i.e. one object violates the safety ellipsoid of the other one, are added to the final list of close conjunctions. In the original algorithm even conjunctions with a distance larger than the maximum axis of the safety ellipsoid were inserted into this temporary list. Those conjunctions later on are assigned with a constant negative penetration factor and therefore discarded at the final selection stage. By eliminating those conjunctions as early as possible we reduced the number of conjunctions that need to be analysed drastically, as only very few conjunctions are closer than the maximum axis of the safety ellipsoid. This optimization will therefore have the biggest impact on the calculation of the penetration factor and the conjunction definition. The second optimization concerned the FindTCA (Time of closest approach) function. After completing the actual computation for each potential pair (parallelized with OpenMP), the function loops again over all potential pairs using a single thread to check for error flags. However before checking if an error happened, the program computes a string to print the current time as preparation for the actual error message. This is done for every potential pair where a conjunction has been detected. This small mistake in the error handling had a big impact on the overall performance of this function.

4.3. The GPGPU CAN system

Fig. 5 shows the time line of the adapted system, with both the CPU and the GPU and the the data transfer between them. Minimizing the overhead produced by these allocations and transfers turned out to be crucial, as it can prevent the whole performance improvement achieved by the kernels. For example the number of potential pairs to analyse and the number of state vectors in the ephemeris varies from epoch to epoch. To prevent unnecessary and expensive reallocations, all variables whose size is related to those changing values are allocated with a margin, i.e. only in case of an unexpectedly big increase of potential pairs or state vectors reallocation is necessary. The following parts of the CAN system have been adapted. Please note that the algorithms described are all adopted (and adapted) from the original CAN system and therefore please read [6, 4, 5] for more detailed descriptions.

Interpolating for the Smart Sieve and Linear Search: At any given time during the programs execution memory contains the ephemeris data for several epochs, including the current one. This ephemeris data is stored as set of position-velocity 6-vectors at regular time intervals for each object. Using Lagrange polynomials the interpolation method can approximate a state vector for every point in time within the boundaries of the ephemeris data in memory. The interpolation for the Smart Sieve takes place once per epoch before the Smart Sieve to calculate the exact state vectors at the beginning of the current epoch. The interpolation for the Linear Search on the other hand is called for every time step of the Linear Search algorithm, the number of invocations depends on the length of those time steps (e.g. $T_{prediction} = 8$ days, $T_{epoch} = 180s$, $T_{timestep} = 6s$).

In the original CAN system OpenMP is used to parallelize the loop that runs over all the objects, which typically uses as many threads as CPU cores available. C CUDA on the other hand allows us to run one thread for each object, i.e. each thread executes one iteration of the loop. However providing the necessary data for the computations on the device is more demanding. In order to interpolate the current state vectors the method needs access to the ephemeris data, which is located in the host memory, i.e. we need to copy it to device memory. GPU memory is limited compared to main memory and therefore our first approach was to identify all data required to interpolate all state vectors needed in the current epoch and then load it to device memory. The main

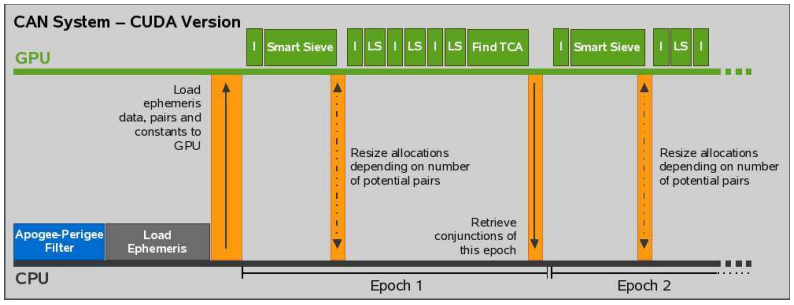

Figure 5. Time line of the GPGPU version of CAN.

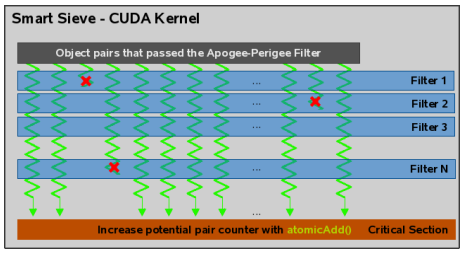

Figure 6. GPGPU version of the Smart Sieve.

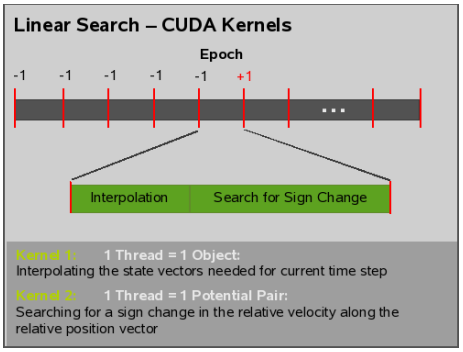

Figure 7. GPGPU version of the Linear Search.

advantage of this approach is the minimal device memory consumption. A first performance analysis however revealed a considerable overhead produced by this data transfer, as it requires thousands of small allocations and transfers instead of a few big ones. To counter that we decided to load the ephemeris data to device memory for several epochs at once, i.e. right after we loaded it from file or database to host memory. This way we extended the partial loading mechanism of the original system to device memory. A further performance analysis backed this second approach. On the downside the upper limit of epochs that can be loaded to memory at once decreased due to the GPUs limited memory.

Smart Sieve: At the beginning of each epoch, this function filters all pairs by analysing the relative position and velocity using different filter steps. Only pairs that pass all filter steps are added to the list of potential conjunction pairs for the current epoch. In the original CAN system OpenMP is used to run the filtering of the pairs in parallel threads. For 10000 objects, the number of pairs that pass the apogee-perigee filter goes into the millions, which means we could use millions of parallel threads.

The CUDA kernel we developed launches one thread for each pair that passed the Apogee-Perigee Filter. Each thread then has to pass a series of filters. As soon as a pair fails one filter, the thread terminates and the pair is eliminated for the current epoch. All threads execute the same instructions in the same order, which makes this approach ideal for the SIMT architecture of the GPU. All active threads in a warp are guaranteed to execute the same instruction, because the ones that take a different branch (i.e. do not pass a filter) terminate immediately. Once a thread passes all the filters, it enters the critical section. A critical section, in parallel programming terms, is a section where only one thread at a time is allowed to enter, because a shared resource is accessed and/or modified. In the Smart Sieve this shared resource is a counter that registers the number of pairs that passed this filter. Having a critical section like this one is generally very expensive, our first implementation was therefore trying to avoid it by having a single CPU thread summing up the local counter of each GPU thread. This solution performed better than the original implementation, however after the first detailed performance analysis we noticed that only around 0.2% of all pairs actually pass all filters. Therefore only a very small number of threads is competing for the shared resource. Our final implementation is using the atomicAdd operation provided by CUDA to guarantee mutual exclusion, which proved to be considerably faster than the previous approach.

Linear Search: After the two filtering steps (Apogee-Perigee Filter and Smart Sieve) this is the first actual conjunction analysis step. The epoch is divided into small time steps and a linear search is conducted. In order to identify a conjunction, the sign of the relative velocity along the relative position vector is computed for each time step and compared with the sign of the previous time step. A change of the sign from negative to positive implies that the objects have been approaching each other and passed the point of closest approach. Thus a conjunction has been detected between those objects.

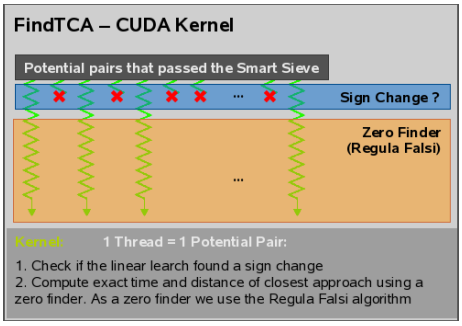

Figure 8. GPGPU version of FindTCA function.

For each time step there are two distinct tasks, the first one is to interpolate the state vectors for all objects for the current time step, the second one is the computation and comparison of the sign. Therefore two different CUDA kernels are executed for every time step. As we described before, the interpolation kernel launches one thread per object and uses Lagrange polynomials to interpolate the current state vector from the ephemeris data in GPU memory. The second kernel launches one thread for each potential pair. Each thread computes the sign and compares it with the sign of the previous time step, which still resides in GPU memory from the previous kernel launch. Although this function requires a large number of CUDA kernel calls, there is almost no data transfer to and from GPU memory necessary in between kernel calls, as the ephemeris data is already present in GPU memory as well as the interpolated state vectors.

FindTCA (Time of Closest Approach): Once the time steps with the conjunctions are identified by the Linear Search kernel, this functions determines the exact distance and time of closest approach using a zero finder algorithm. The Regula Falsi algorithm that is used as zero finder is a iterative algorithm that requires repetitive calculation of the derivative of the relative velocity in order to find the exact time of the zero crossing, i.e. the time where the sign changes from negative to positive.

The function is implemented as a single CUDA kernel and launches one thread for each potential pair. If there has been a sign change detected for a pair the thread enters the zero finder, otherwise it terminates. The Regula Falsi algorithm requires that the ephemeris data is available in GPU memory for the computation of the derivative of the relative velocity function. Our first approach was similar to the one we started from in the interpolation kernel. The ephemeris data required for the computation of the derivatives is identified and loaded to GPU memory. But like in the interpolation kernel this approach produced a considerable overhead and with the extension of the partial loading mechanism to GPU memory it became obsolete. Once the Regula Falsi algorithm completes its final iteration, the exact distance and time of closest approach is loaded back from GPU memory to main memory where the collision risk is computed and the close conjunctions are inserted in the final conjunction list.

Table 1. Performance gain for a catalogue with 15'000 objects for 8 days

Original CAN system	42 min	-	-
Optimized CAN system	22.2 min	-47.1%	1.9x
GPGPU CAN system	4.5 min s	-89.3%	9.4x

5. RESULTS

We elaborated and implemented the following improvements to the CAN system:

- Adaptation of several core functions of the conjunction prediction algorithm for execution on the GPU (filtering, ephemeris data interpolation, conjunction detection)
- Non-GPU related improvements in the algorithm (conjunction definition, error checking)

All performance measurements have been performed on the same OpenSUSE Linux machine with a Intel Xenon CPU (E5620 4 Cores @ 2.4 GHz) and 6GB of memory. Additionally we equipped the machine with a CUDA-enabled GPU, a NVIDIA Geforce GTX 580 (1.5 GB memory, 512 CUDA cores, compute capability 2.0). The following compilers were used: Intel Compiler (Fortran, Linking), NVIDIA C CUDA Compiler, GCC.

Thanks to those two improvements we reduced the runtime of the computational part of the CAN system by up to 89%, i.e. it was reduced from 42 min to 5 min. Tab. 1 summarizes the runtime of the computational part of an all vs. all conjunction prediction task with 15'000 objects and a prediction period of 8 days. The comparison includes three different versions of the CAN system. The original CAN system, a version that only includes the algorithm optimizations and the GPGPU version including both the algorithm optimization and the CUDA kernels.

The complete performance measurement included different catalogue sizes ranging from 300 to 15'000 objects and from a one day prediction period to a eight day prediction period. We observed the expected quadratic growth with respect to the number of objects and a linear growth with respect to the length of the prediction period for all versions, also described in [4].

The algorithm optimizations resulted in a performance gain with a constant factor of around 40% for all catalogue sizes. This particular performance gain depends a lot on the distribution of the objects in the catalogue, as the optimizations are related to the number of conjunctions and number of potential pairs (pairs that passed the Smart Sieve). Therefore it is possible that this factor changes for different prediction tasks, e.g. including only

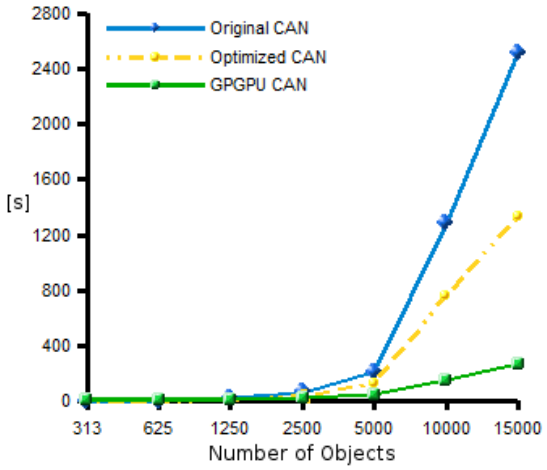

Figure 9. Conjunction Analysis for 8 days using different catalogue sizes

objects from one orbital belt, which could lead to more conjunctions compared to the number of objects.

The GPGPU version outperformed both other versions by several orders of magnitude for big catalogues. The factor of the performance gain is increasing with the catalogue size and therefore the best performance is shown with our biggest catalogue of 15'000 objects and we expect that this trend will continue. For small catalogues, the small number of parallel threads and the overhead produced by transferring data to GPU memory lead to a significant decrease of the performance. To be more precise, for 1250 objects the GPGPU version is still faster than the original CAN system, but slower than the optimized version. For even smaller catalogue sizes the GPGPU version is up to 4 times slower than the original version.

The more detailed performance analysis (fig. 10) shows which functions of the algorithm profited the most from the GPU and the algorithm optimizations.

The algorithm optimization described in section 4.2 reduced the time spent for conjunction definition and the calculation of the penetration factor to near zero. The optimization of the error checking leads to a significant performance gain in the FindTCA function. However the comparison between the original CAN system and the optimized versions should be treated with caution. The original is written in Fortran and compiled by the Intel Compiler (IFORT) and the optimized one is written in C, compiled with NVCC (which invokes GCC) and linked to Fortran code using the Intel Compiler. Furthermore both versions use slightly different OpenMP libraries. We believe that the language and compilers account for most differences aside from the effects of the algorithm optimizations mentioned above. However the performance gain in the Smart Sieve function prompts questions, as there is no significant difference in the code that could explain this 30% improvement. A possible explanation could be that the synchronization between the threads, that is required for the critical section of this function, is handled differently in both OpenMP libraries.

The functions that profited the most from their CUDA kernel are Linear Search (10 times faster) and the Smart Sieve (12.5 times faster). The FindTCA and the interpolation function improved only by a factor between 2.5 and 3.5. The former functions require mostly branchless and simple computations, which is ideal for the SIMT architecture of the GPU. As described in section 3 this architecture reaches optimal performance only if all threads in a warp execute the same instruction at the same time. The latter functions both require more complex computations and FindTCA even executes an iterative algorithm (Regula Falsi) in every thread, which creates a larger variety of instructions that have to be executed at a given point in time. Our efforts to reduce the time spent on transferring data to and from GPU memory clearly manifest in this measurement, as only a very small fraction of the overall runtime of the GPGPU version is spent for CUDA memory operations.

6. FUTURE WORK

The computational part of the CAN system was reduced significantly, hence the distribution of the overall runtime changed drastically. The I/O, the reading and writing of ephemeris data and conjunction information to and from database now clearly outweighs the computational part. The GPU has no direct access to the hard drive or the network adapter of its machine and therefore I/O operations can not (yet) be adapted to be executed on the GPU. However could one use the additional computational power of the GPU to recompute the required data instead of loading it from database. In the CAN system, the ephemeris data required for the computations consists of a large number of state vectors for each object that have been computed from a single state vector using orbit propagation. By recomputing the required state vectors at runtime using the GPU the CAN system could overcome the I/O bottleneck.

7. CONCLUSION

We presented important performance gains for the CAN system regarding all vs. all conjunction predictions for large catalogues, first by harvesting the power of hundreds of GPU cores and second by improving the algorithms. The improvement of the computational part of the original CAN system to the GPGPU version of CAN clearly shows the viability of our approach. With the computational part improved by up to 89%, the I/O is now dominating the runtime of a conjunction prediction task. However considering very small and very large catalogues, two major limitations of the GPGPU approach emerge. For small catalogue sizes the overhead of loading data to and from the GPU and the small number of parallel threads produces a significant overhead, enough

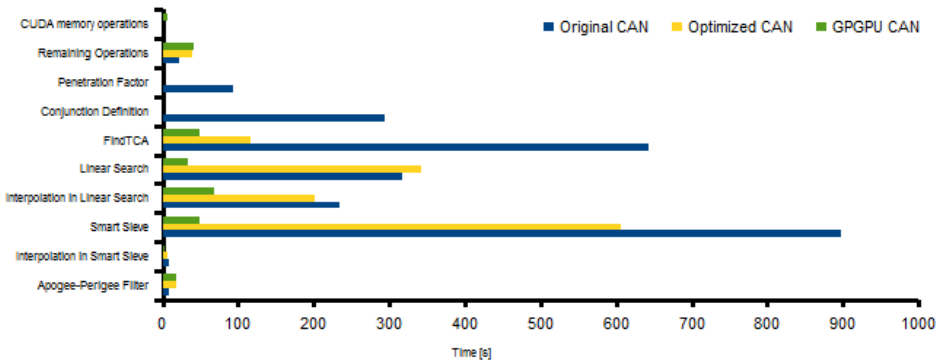

Figure 10. Detailed performance analysis for 15'000 objects for 8 days

to completely negate the gain achieved by the faster computations and decrease the performance by several orders of magnitude. For very large catalogues, larger than what is currently available, calculations show that the memory consumption will require significant changes in the conjunction analysis algorithm. The current design permits catalogue sizes of up to 15'000 for the GPGPU version, which is a slightly lower boundary than the original CAN system. However the memory consumption for both versions with the current algorithm design is growing quadratically and will exceed available GPU memory as well as the main memory. Furthermore we identified other elements of the CPS system that could benefit from GPGPU, e.g. online orbit propagation instead of loading the ephemeris data from database or the calculation of the collision risk. We expect that the former would reduce or even remove the bottleneck produced by the expensive I/O operations.

ACKNOWLEDGMENTS

The authors would like to thank Diego Escobar, one of the developers of the CAN system, for his valuable comments and insights into the system and for providing tools and data for testing. Furthermore we would like to thank the SSA team for their general support, but especially for their help with acquiring and setting up the test environment.

REFERENCES

1. NVIDIA, (March 2013). CUDA Programming Guide, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
2. NVIDIA, (March 2013). CUDA best practice guide, `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`
3. NVIDIA, (March 2013). CUDA Architecture overview, `http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf`
4. Escobar, D., Àgueda, A., Martin, L. and Martinez, F. M., (2011). Predicting Collision Risk for European SSA System with closeap, Proceedings of the 22nd International Symposium on Space Flight Dynamics, Brazil
5. Escobar, D., Àgueda, A., Martin, L. and Martinez, F. M., (2011). Efficient all vs. all collision risk analyses, Proceedings of the 22nd International Symposium on Space Flight Dynamics, Brazil
6. Escobar, D., Àgueda. A., Martnez, F. M. and Garca, P., (2010). Implementation of conjunction assessment algorithms in modern space mechanic systems, Proceedings of the 4th International Conference in Astrodynamics Tools and Techniques, Madrid, Spain
7. OpenMP Architecture Review Board, (March 2013). OpenMP Specifications, `http://openmp.org/wp/openmp-specifications/`
8. European Space Agency Council. Declaration on the Space Situational Awareness (SSA) Preparatory Programme, ESA/C(2008)192, Paris, France 8th December 2008.
9. H. Klinkrad, (2006). Space Debris: models and risk analysis, Springer