A Strategy to Determine Whether to Use GPU for a Satellite Mission Scheduling Algorithm

By Soojeon LEE, Byoung-Sun LEE and Jaehoon KIM

Satellite System Research Team, Electronics and Telecommunications Research Institute, Daejeon, South Korea

(Received June 9th, 2011)

As the first Korean multi-mission geostationary satellite, Chollian was launched on June 27, 2010. Chollian is being successfully controlled using a satellite ground control system (SGCS) developed by ETRI. A mission planning subsystem (MPS) in SGCS gathers mission requests from users, performs complex mission scheduling, and generates a conflict-free mission schedule. In this paper, we provide an overview of the current mission scheduling algorithms of the Chollian satellite, select three representative constraint checking schemes among these algorithms, and implement new graphics processing unit (GPU)-based constraint checking schemes for the three representative schemes. We compare the performance of the GPU-based and CPU-based constraint checking schemes based on the size of the problem set and the time complexity of the problem. Finally, we suggest a strategy to determine whether or not to adopt GPU for a satellite mission scheduling algorithm.

Key Words: Mission Scheduling Algorithm, Mission Planning System, COMS, Chollian Satellite, CUDA, GPU

1. Introduction

As a multi-mission geostationary satellite, Chollian, also called the Communication, Ocean, and Meteorological Satellite (COMS), was launched on June 27, 2010. The Chollian satellite is located at 128.2 degrees East longitude and 36,000 km from the Earth. This makes Korea the tenth country in the world to develop a geostationary communications satellite, which will operate over the next seven years. The Chollian satellite has three different payloads for three different purposes: satellite communications, ocean observations and meteorological observations. Especially for the satellite broadcasting and telecommunications, there has been various research conducted and ETRI developed the Ka-band communications payload for the Chollian satellite.

For the operation of the Chollian satellite, several ground segments cooperate as shown in Fig. 1. Users from the Communications Test Earth Station (CTES), the Korea Ocean Satellite Center (KOSC), and the Meteorological Satellite Center (MSC) submit mission requests to the satellite ground control system (SGCS). The image data acquisition and control systems (IDACSs) in KOSC and MSC receive raw data from the satellite and perform image preprocessing to generate Level1B data.

Even if each user provides a perfect conflict-free mission request for his/her own organization, conflicts can frequently occur after mixing the requests received from different organizations. In one of the examples, if meteorological imaging and oceanic imaging are both executed at the same time, degradation in image quality may occur in a meteorological image. To prevent these problems, Chollian-specific mission scheduling algorithms are required.
constraint checking schemes, which are able to be applied for other satellites’ mission scheduling, as well. Constraint checking schemes used for the Chollian satellite’s mission scheduling algorithms are categorized into three groups. For a representative scheme in each category, we implement a GPU-based version and compare the performance with a CPU-based one. Finally, we suggest a simple but efficient strategy to determine whether or not to use GPU for a satellite mission scheduling algorithm.

2. Mission Planning Overview

The SGCS of the Chollian satellite enables the satellite operator to execute the satellite missions and control the satellite. The SGCS consists of five subsystems: telemetry, tracking and command (TTC), real-time operations subsystem (ROS), flight dynamics subsystem (FDS), and COMS simulator subsystem (CSS), as shown in Fig. 2. The mission scheduling algorithms in this paper are implemented in an MPS.

2.1. Meteorological and oceanic missions

2.1.1. Missions via meteorological imager

There are three meteorological imager (MI) observation modes: global, regional and local. The global mode includes full disk (FD) imaging that covers the entire Earth. FD imaging normally takes less than 1,620 s. The regional mode includes the Asia-Pacific Northern Hemisphere (APNH), the Extended Northern Hemisphere (ENH), and the Limited Southern Hemisphere (LSH) areas. The local mode includes the Local Area (LA) imaging, which covers a randomly selectable area in the FD boundary.

2.1.2. Missions via geostationary ocean color imager

A geostationary ocean color imager (GOCI) takes oceanic images around the Korean peninsula. A GOCI image can include a maximum of 16 slots, and contiguous slots have overlapping areas.

2.2. Mission scheduling steps

2.2.1. MI and GOCI algorithms

For MI mission requests received from the MSC, image duration calculation, scan coordinate conversion, and proportional command generation are performed by the MI algorithm. For GOCI mission requests received from KOSC, the displacement angle of the mirror pointing mechanism is calculated by the GOCI algorithm.

2.2.2. Constraint check

Constraints are checked using pre-defined Chollian-specific relation rules such as exclusion, inclusion and predecessor-successor relationships among missions, including event information and maneuver requests.

2.2.3. Priority check

If missions that have an exclusion relation and different priorities overlap with each other, the mission having lower priority is always discarded based on the priority rules.

3. Analysis of the Constraint Checking Schemes

3.1. Categories

The constraint checking schemes of the Chollian satellite can be categorized as shown in Table 1.

3.1.1. MI image properties

There are two regulations regarding the MI imaging itself. First, the imaging duration of each observation area must be smaller than its maximum duration limit. Second, the imaging boundary of each observation area must be within its maximum boundary limit. These two regulation schemes are called CheckMIMaxDuration and CheckMIImageBoundary.

3.1.2. Overlap of missions

There are two general rules dealing with the overlapping of missions: exclusion and inclusion. The former is a regulation stating that missions that have an exclusion relation must not be executed simultaneously, while the latter states that mission A must be executed within mission B’s time window if there is an inclusion relationship, A ⊆ B, between them. These two regulation schemes are called CheckExclusion and CheckInclusion.

3.1.3. Predecessor-successor relations

Four rules exist depending on the predecessor-successor relations. The first rule is about the sequence of missions. If there is a sequence rule A → B → C, mission A must be followed by mission B, while mission B must be followed.
by mission C. There must not be any missions between them. The second rule regulates that some specific sequences must not be established. This is the opposite of the first rule. For example, the rule \(^{(A \rightarrow B \rightarrow C)}\) means that the sequence \(A \rightarrow B \rightarrow C\) must not be allowed. The third and fourth rules are about the time gap between two missions. If mission A precedes mission B, the spacing time between the end time of mission A and the start time of mission B must be smaller than its allowable maximum limit in the former, while it must be larger than its allowable minimum limit in the latter. The four regulation schemes are called CheckSequence, CheckNonSequence, CheckMaxTimeGap and CheckMinTimeGap.

3.2. Selection of representative constraint checking schemes

Among the constraint checking schemes, we select one representative scheme in each category: CheckMIMaxDuration, CheckExclusion and CheckNonSequence. The reason these are chosen is that they are the most frequently used in this category and generate most of the conflict messages during normal mission planning scenarios.

For the selected three schemes, which are CPU-based and are currently used for the Chollian satellite mission planning, we implement corresponding GPU-based schemes. We then compare the performance of the CPU- and GPU-based schemes.

3.3. Analysis of the CPU-based constraint checking schemes

3.3.1. CheckMIMaxDuration

CheckMIMaxDuration can be described as shown in Fig. 3. MissionRequest and mi indicate a mission request and MI mission, respectively. For all missions in a mission request, it checks whether the duration of an MI mission exceeds its maximum limit. If it does, it sets the mission’s conflict flag to true. Let \(N\) be the number of missions in the mission request. The time complexity of this scheme is then \(O(N)\).

3.3.2. CheckExclusion

Figure 4 shows the pseudo code of CheckExclusion. An exclusion rule consists of a pair of mission IDs (i.e., source ID and destination ID). For each of the exclusion rules, it loops for all missions in a mission request. It then checks if the rule’s source ID is equal to a mission’s ID. If it is, that mission is called src. The scheme then obtains all of the overlapping src missions and searches those having an ID equal to the exclusion rule’s destination ID by looping over the overlapping missions. If found, the mission is called dest, and the conflict flags of the src and dest are set to true.

Let \(R\) be the number of exclusion rules and \(N\) be the number of missions in the mission request. In Fig. 4, GetOverlapMissions() consumes \(O(N)\) time, and thus the time complexity of this scheme is \(O(RN^2)\). Note that as of May 2011 there are 48 exclusion rules in the Chollian MPS.

3.3.3. CheckNonSequence

Figure 5 shows the pseudo code of CheckNonSequence. A non-sequence rule consists of a pair or triplet of mission IDs (i.e., (first ID, second ID) or (first ID, second ID, third ID)). For each of the non-sequence rules, it loops for all of the missions in the mission request. It checks if the rule’s first ID is equal to a mission’s ID. If it is, that mission is called first. Next, it obtains the next mission of first, which is called second. If second is null, it means that first is the last mission in the mission request, so it breaks the loop. Otherwise, it compares the second ID of the rule and the ID of second. If second is null, it means that first is the last mission in the mission request, so it breaks the loop. Otherwise, it compares the second ID of the rule and the ID of second. If they are the same, it checks the length of the rule. If the length of the rule is two, the conflict flags of first and second are set to true and it breaks the loop. If the length of the rule is three, it obtains the next mission of second, which is called third. If third is null, it means that second is the last mission in the mission request, so it breaks the loop. Otherwise, it compares the third ID of the rule and the ID of third. If they are the same, the conflict flags of first, second and third are set to true and it breaks the loop.
Let $R$ be the number of non-sequence rules and $N$ be the number of missions in the mission request. In Fig. 5, `GetNextMission()` consumes $O(1)$ time, and thus the time complexity of this scheme is $O(RN)$. Note that as of May 2011, there are 11 non-sequence rules in the Chollian MPS.

4. Analysis of the GPU-based Constraint Checking Schemes

To utilize the power of the GPU, we adopt Nvidia’s Compute Unified Device Architecture (CUDA). Figure 6 shows a simplified architecture of Nvidia GeForce GTS 250. There are 16 multiprocessors and each multiprocessor contains eight cores. Thus, in total 128 cores can operate at the same time. In a multiprocessor, the instruction unit makes each core execute the same instruction and CUDA threads running on each core can share data through a shared memory.

To compare the performance of CPU-based conflict checking schemes, we implement the GPU versions, `cudaCheckMIMaxDuration`, `cudaCheckExclusion` and `cudaCheckNonSequence`.

The Chollian satellite’s MPS was developed using C# language on a .NET framework. However, the CUDA programming model basically supports C language, even though there have been other efforts, e.g., GPU.NET or CUDA.NET to enable CUDA to run on the .NET framework. These efforts are mostly commercial and/or have not been proven well in the industry thus far. Therefore, we compile C style CUDA codes with an Nvidia C compiler (NVCC) and generate .dll files. The compiled .dll files can be called and used in MPS via the C#’s attribute, `DLLImport`.

For simplicity, we divide GPU-based constraint checking schemes into three parts: C# caller, CUDA host and CUDA device. For instance, the C# caller calls the CUDA host through the CUDA wrapper. The arrays are copied into the device memory and, as in Fig. 7(b), the CUDA kernel `CUDACheckMIMaxDuration` is invoked. The thread block size `threadsPerBlock` is set to 256 because it is a common choice, as proposed by Ref. 18. In a kernel invocation, $(\text{threadsPerBlock} \times \text{blocksPerGrid})$ CUDA threads are executed. As shown in Fig. 7(c), each CUDA thread checks whether a mission with index $i$ has a conflict, and sets its conflict flag to true if it does.

4.1. `cudaCheckMIMaxDuration`

`cudaCheckMIMaxDuration` can be described as shown in Fig. 7. As in Fig. 7(a), the durations and IDs of all missions in the mission request are stored into `dur_array` and `ID_array`. Using its parameters such as array count and conflict flag array, the C# caller calls the CUDA host through the CUDA wrapper. The arrays are copied into the device memory and, as in Fig. 7(b), the CUDA kernel `CUDACheckMIMaxDuration` is invoked. The thread block size `threadsPerBlock` is set to 256 because it is a common choice, as proposed by Ref. 18. In a kernel invocation, $(\text{threadsPerBlock} \times \text{blocksPerGrid})$ CUDA threads are executed. As shown in Fig. 7(c), each CUDA thread checks whether a mission with index $i$ has a conflict, and sets its conflict flag to true if it does.

4.2. `cudaCheckExclusion`

`cudaCheckExclusion` can be described as in Fig. 8. As shown in Fig. 8(a), the start times, end times and IDs of all missions in the mission request are stored into `start_time_array`, `end_time_array` and `ID_array`. Likewise, all exclusion rules’ source IDs and destination IDs are stored into corresponding arrays. Then, the C# caller calls the CUDA host through the CUDA wrapper. The arrays are copied into the device memory as shown in Fig. 8(b). Compared to the `cudaCheckMIMaxDuration`, `cudaCheckExclusion` requires many more CUDA threads and blocks to run, as the time complexity of this problem when using a CPU is basically $O(RN^2)$, as shown in Fig. 4. Even though, theoretically, $4,294,967,296$ (65,535 * 65,535) blocks can be used by a kernel, we cannot use them maximally due to several constraints, such as the total amount of available memory on a device.

To schedule a normal Chollian satellite mission request, the number of CUDA threads or number of blocks frequently exceeds the above systematic limits, which leads to a pro-
gram crash. To prevent this abnormal situation, we divide one large kernel invocation into pieces. Thus, the kernel CUDACheckExclusion is invoked multiple times through looping depending on the amount of calculation. We set the maximum number of blocks to 65,535, and the maximum number of threads to 16,776,960 (65,535/C3/256), for a kernel invocation even though we can obtain a higher performance using more than 65,535 blocks.

loop cnt represents the number of necessary kernel invocations. In Fig. 8(c), rule index i, mission index j, and mission k in the mission request are acquired, and mission j and mission k’s conflict flags are set to true if a conflict exists between them.

5. Performance Evaluation

We compare the execution times of CPU-based conflict checking schemes with GPU-based ones. Table 2 shows the experiment environment in terms of hardware and software.

4.3. cudaCheckNonSequence

cudaCheckNonSequence can be described as shown in Fig. 9. As shown in Fig. 9(a), the IDs of all missions in the mission request are stored into the ID_array. Likewise, all first, second and third IDs of the non-sequence rules are stored into corresponding rule arrays. Note that the third ID can be null if a rule is not a triplet but a pair. Then, the C# caller calls the CUDA host through the CUDA wrapper. The arrays are copied into the device memory and cudaCheckNonSequence is invoked as shown in Fig. 9(b).

In Fig. 9(c), rule index i, and mission index j in the mission request are acquired, and we check whether a conflict exists in the CUDA thread. If mission j and mission j+1 have a conflict, the conflict flags of the two missions are set to true. If mission j, mission j+1 and mission j+2 have a conflict, the conflict flags of the three missions are set to true.
There are a number of general ways\(^{17}\) to further increase the performance of a GPU-based approach: using shared memory instead of global memory, avoiding a bank conflict in shared memory, coalesced global memory access, less use of conditional branches and so on.

The nature of the Chollian conflict-checking schemes, however, requires storing the conflict results within multiple if-statements, which prevents the parallel execution of threads in a warp and causes a huge delay. Figure 10 describes the execution flow of CUDA threads in a multiprocessor and explains why conditional branches cause a delay. Here, 32 CUDA threads, of which a warp consists, should execute the same instruction. A multiprocessor has only eight cores, so 32 CUDA threads on a warp cannot execute simultaneously. Thus, a unit of eight CUDA threads executes serially in a warp. For instance, if eight CUDA threads (T1–T8) in Warp 1 have conditional branches, it is possible that one CUDA thread (T1) goes into if-clause, while the others (T2–T8) go into else-clause. In this case, all the others (T2–T8) in else-clause should wait until the one (T1) in if-clause finishes because the cores should execute the same instruction. Thus parallel execution of threads in a warp cannot be accomplished.

Despite this fundamental disadvantage, GPU-based conflict-checking schemes tend to show a better performance if the size of the problem set becomes larger, or the time complexity of the CPU-based schemes is higher.

### 5.1. CheckMIMaxDuration vs. cudaCheckMIMaxDuration

The maximum duration limits of the FD, APNH, ENH, LSH and LA are 1,620, 243, 742, 396 and 60, respectively. If the duration of an MI mission is larger than its limit, an alarm must be sent to the mission operator.

Only MI missions are affected by this constraint checking scheme. Thus, we use only MI mission requests to compare the performance. The MI mission request used, which was actually used during the Chollian satellite’s in-orbit test (IOT) phase, is for the day of Sept. 21, 2010, and consists of 302 missions in total, i.e., 49 MI sequences, 49 block body calibrations and 204 MI missions (7 FDs, 37 APNHs, 37 ENHs, 86 LAs, and 37 LSHs).\(^{19}\)

To repeat the above one-day MI mission request, we generate multiple-day MI mission requests, i.e., for 2, 4, 8, 16, 32, 64, 128, 256, 512, 1,024 days. For instance, the 1,024-day MI mission request contains 309,248 (1,024\(^3\)) missions and covers from Sept. 21, 2011 to July 10, 2013.

There are three reasons why we generate a multiple-day mission request by repeating a one-day mission request instead of using a real multiple-day request. First, a mission request during an IOT is not always similar to that during normal operation. Various experimental tests are performed during an IOT, and thus a mission request on a particular day can be quite uncommon and different with that of another day. This can cause a bias in the experiment. Second, the mission request for the day of Sept. 21, 2010 was quite close to a normal operation scenario.

Thus, we can acquire a near-real multiple-day mission request by repeating it. Third, a sufficient number of real multiple-day mission requests have not been accumulated as of May 2011, and thus we cannot obtain real large multiple-day (512 or 1,024 days) mission requests.

Figure 11 shows the execution times of CheckMIMaxDuration and cudaCheckMIMaxDuration. When the size of a mission request is large, e.g., from 512 to 1,024, cudaCheckMIMaxDuration outperforms CheckMIMaxDuration due to its parallelism; 128 CUDA cores simultaneously execute the device code described in Fig. 6. However, when the size of a mission request is relatively small, e.g., from 1 to 128, CheckMIMaxDuration shows a shorter execution time than cudaCheckMIMaxDuration. With the same size mission requests, cudaCheckMIMaxDuration shows rather constant execution times. Why is there no advantage of parallelism in this case? It is because of the default CUDA setup overhead. In a CUDA kernel invocation, CUDA setup is necessary to initialize the CUDA context on the GPU, allocate memory and release the CUDA context. Even when a...
CUDA kernel has nothing to do, about 60–70 ms of CUDA setup overhead exists in this case. It means that a CUDA kernel invocation for a small calculation is not an efficient way of solving a problem.

5.2. CheckExclusion vs. cudaCheckExclusion

For the MI mission request described in section 5.1, we add other types of mission requests. They were also used during the Chollian satellite’s IOT phase for Sept. 21, 2010. Thus, the mission request we use consists of 346 missions in total, i.e., 49 MI sequences, 49 block body calibrations, 204 MI missions (7 FDs, 37 APNHs, 37 ENHs, 86 LAs, and 37 LSHs), 8 GOCI sequences, 8 GOCI shutter opens, 8 GOCI shutter closes, 8 GOCI missions, 9 events (1 apogee crossing, 1 perigee crossing, 2 nodal crossings, 1 eclipse by the Earth, 2 sensor intrusions by the Sun, 2 sensor intrusions by the Moon), 1 North-South station keeping, and 2 wheel-offloadings.14)

By repeating the above one-day mission request, we generate multiple-day mission requests, i.e., for 2, 4, 8, 16, 32, 64, 128, 256, 512, 1,024 days, as described in section 4.

Figure 12(a) shows the execution times of CheckExclusion and cudaCheckExclusion. cudaCheckExclusion shows a shorter execution time than CheckExclusion. For example, when the size of a mission request is eight, the execution times of CheckExclusion and cudaCheckExclusion are 7,285 and 904 s, respectively; cudaCheckExclusion is nearly 8.1-times faster than CheckExclusion.

Figure 12(b) shows the execution time ratio of CheckExclusion to cudaCheckExclusion. In each case, cudaCheckExclusion outperforms CheckExclusion. When the size of a mission request is one, cudaCheckExclusion is only 1.8-times faster than CheckExclusion, but converges to 4.5-times faster in a 1,024-day mission request.

Figure 12(c) shows the correlation between the number of kernel invocations and the execution time of cudaCheckExclusion per kernel invocation. The former and latter are expressed with red bars and blue dots, respectively. As the number of missions $N$ in a mission request becomes larger, the number of kernel invocations increases in proportion to $N^2$. The execution time per kernel invocation is converged to a certain level, i.e., 85 ms.

5.3. CheckNonSequence vs. cudaCheckNonSequence

We experimented on the same mission requests described in section 5.2. Figure 13 compares the execution times of CheckNonSequence and cudaCheckNonSequence.

When the size of a mission request is less than or equal to 32, CheckNonSequence shows a shorter execution time than cudaCheckNonSequence. However, if the size of a mission request is larger than 32, cudaCheckNonSequence is faster than CheckNonSequence. This trend is similar to that described in section 5.1. In both cases, as shown in Figs. 11 and 13, GPU-based schemes show rather constant execution times with small problem sets, and outperform the CPU-based schemes in large problem sets. This is because the time complexity of the problem itself is basically the same as $O(N)$ for both schemes.

In summary, Fig. 14 suggests a strategy to determine whether or not to adopt GPU for a satellite mission scheduling algorithm. If the time complexity of a problem is high, using GPU is likely to be a good choice. Otherwise, consider the size of the problem set; use GPU only if the problem set is large. This strategy can be an efficient guide while determining the adoption of GPU in the satellite mission scheduling areas.
Stop

Start

Analyse

the size

of problem set

Is problem

set large?

Adopt GPU

Refuse GPU

Analyse
time complexity

Is time

complexity high?

Fig. 14. A strategy determining whether or not to adopt GPU for a mission scheduling algorithm.

6. Conclusion

We implemented GPU-based conflict checking schemes and compared their execution times with those of CPU-based schemes used for the Chollian satellite. A performance evaluation showed that execution time can be reduced tremendously through using a GPU in some cases. Even though the mandatory use of multiple conditional branches in GPU-based conflict checking schemes caused a significant delay, one of the GPU-based schemes was maximally 8.1-times faster than its corresponding CPU-based version.

In general, if the time complexity of a problem is less than or equal to \( O(N) \), the benefit of a GPU-based scheme over a CPU-based scheme increases as the size of the problem set increases. If the time complexity of a problem is larger than \( O(N) \), the benefit of a GPU-based scheme over a CPU-based scheme converges to a certain level if the size of the problem set is large enough. On the contrary, if the problem set is small, adoption of a GPU may not be an attractive solution. Thus, when we adopt the power of a GPU into very critical areas such as satellite control, a detailed analysis regarding the size of the problem set, time complexity of the problem, and programming/maintenance cost must be performed a priori.

The next Korean GEO satellite projects after the Chollian will require much larger mission requests and more complex mission scheduling algorithms due to the improved imaging performance of the new satellite imager. Likewise, not only for the mission scheduling, but also for the other parts of satellite ground systems, such as event prediction, image processing, image analysis and weather forecasting, larger amounts of data will be generated and more complex prediction models will be applied. Based on this work, we plan to utilize the power of the GPU as much as possible for upcoming satellite projects.

References

15) CUDA Background, http://www.nvidia.com/object/what_is_cudanew.html

Appendix

For those who are not familiar with the pseudo codes in Figs. 3–5 and 7–9, flowcharts of the CPU- and GPU-based conflict checking schemes are shown in this section.

A. CPU-based schemes

Figure 15 shows the flowcharts of the CPU-based conflict checking schemes. Note that, even though the flowcharts of CheckExclusion and CheckNonSequence look the same, the time complexity of the two is different. When checking whether a rule is violated or not, CheckExclusion requires an internal loop while CheckNonSequence does not. Thus, for CheckExclusion, CheckExclusion and CheckNonSequence, one, three and two loops are required, respectively.

B. GPU-based schemes

Figure 16 shows the flow charts of the GPU-based conflict checking schemes. cudaCheckMIDuration and cudaCheckNonSequence finishes in just one CUDA kernel invocation, thus no loop is required for them. Instead, checking whether a rule is violated is performed in each CUDA thread, in parallel. However, cudaCheckExclusion may require a loop because the time complexity of the problem is sometimes too large to be solved in a CUDA kernel invocation. Thus, the problem is divided into many pieces and each piece is executed in a CUDA kernel with multiple CUDA threads.
Fig. 15. Flowcharts of the CPU-based conflict checking schemes.

Fig. 16. Flowcharts of the GPU-based conflict checking schemes.