INFORMATION
SYSTEMS IN
MANAGEMENT

Information Systems in Management (2017) Vol. 6 (2) 143−153

THE GPU PERFORMANCE IN COORDINATION OF PARALLEL TASKS IN ACCESS TO RESOURCE GROUPS WITHOUT CONFLICTS

MATEUSZ SMOLIŃSKI

Institute of Information Technology, Lodz University of Technology

In high contention environments, with limited number of shared resources, elimination of resource conflicts between tasks processed in parallel is required. Execution of all tasks without resource conflicts can be achieved by preparing a proper overall schedule for all of them. The effective calculation of conflict-free execution plan for tasks provides the conflictless scheduling algorithm that is dedicated to GPU massively parallel processing. The conflictless scheduling algorithm base on rapid resource conflict detection to mutual exclusion of conflicted tasks in access to global resources and is an alternative to other task synchronization methods. This article presents the performance of modern GPU in calculations of adaptive conflictless task schedule. The performance analysis also takes into account all data transfers to and from the GPU memory in various phases of the conflictless task scheduling algorithm.

Keywords: Resource conflict elimination, conflict free task execution, mutual exclusion, deadlock avoidance, cooperative concurrency control, GPU massively parallel processing, SIMD control SISD, GPGPU using OpenCL.

1. Introduction

Modern software development requires parallel task processing to provide high performance. This follows directly from the construction of computers that usually have multiple processing units. Building a multi-process application or multithreaded program requires access control to global resources. To meet the correctness requirements software developers use known concurrency solutions that generally can be classified as competitive or cooperative concurrency [6,7]. Separation of concurrency problems and selection of synchronization mechanisms with their appropriate usage in code is a source of problems in software development. Additionally software developer has to choose fine or coarse grain strategy for synchronization of access to global resources. Using multiple synchronization mechanisms programmer can encounter task starvation or deadlock problem. Together with an increasing number of tasks and global resources increases the difficulty of preparing a correct concurrent program. This problem especially concerns high contention environments i.e. DBMS, OLTP.

As alternative software developer can group resources according to their usage in program (the division into tasks) and use conflictless scheduling to coordinate tasks in access to global resources. The proposed model of parallel task processing frees the programmer from controlling tasks access to global resources to ensure mutual exclusion [4]. Using conflictless scheduling there is no need to use other synchronization mechanism to manage allocation of global resources. The conflictless scheduling can be applied in various task processing environments and is fully responsible for tasks coordination to access groups of global resources without any resource conflicts. Using conflictless scheduling also guarantees no task starvation or deadlock in access to resources group required by task [5]. This universal task coordination approach bases on rapid resource conflict detection and preparation of schedule, which ensures parallel tasks execution without resources conflicts. When one task finishes its execution then other conflicted tasks is started, according to calculated earlier conflictless schedule. All calculations of conflictless schedules need to be performed frequently and as quickly as possible, but not engaging with the task resources.

Therefore an isolated computing environment was proposed, which is single GPU card, to calculate conflictless schedule. The conflictless scheduling concept with massively parallel GPU processing was presented on figure 1. Efficient calculation of conflictless schedule with GPU requires dedicated data structures and algorithm, which will be presented in next chapter. Also example of GPU performance results, obtained using prepared author’s simulator software, will be presented. Also analysis of those results will be discussed in the following sections.

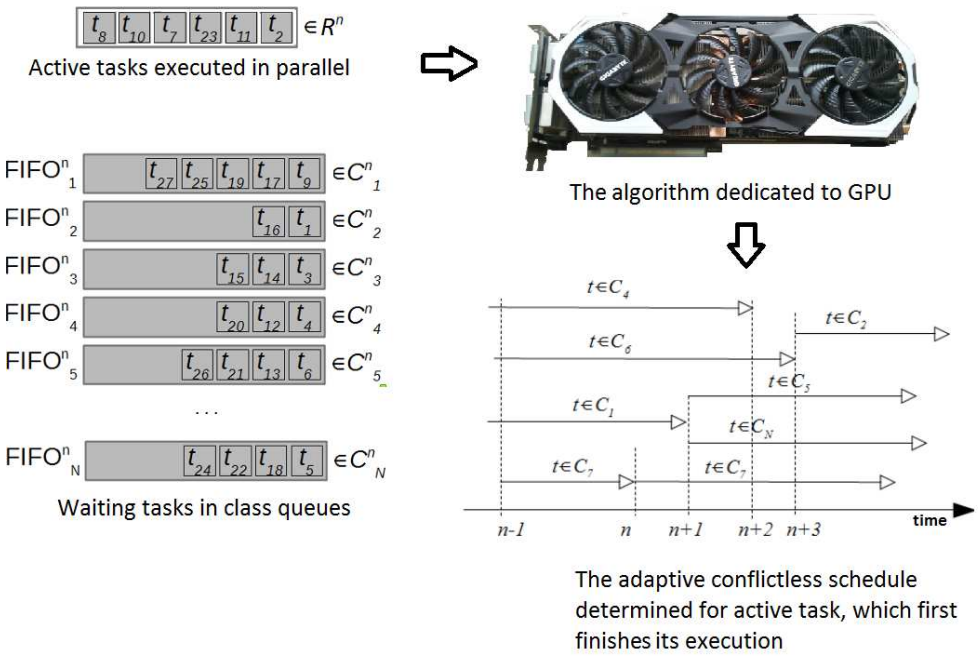

Figure 1. The concept of task coordination in access to global resources using conflictless scheduling

2. The conflictless scheduling

Application of conflictless task scheduling is possible only if all assumptions about environment are fulfilled. The first assumption requires that task processing environment has high contention of resources and the number of global resources is static. High level of contention means that every task has at least one conflict with others in access to limited number of global resources. Second assumption requires that before task execution begin all its global resources has to be known. All required by task resources has to be known in the moment, when task is created. It is worth to mention, that the task execution times do not have to be known and there are no task priorities.

Using binary resource identifiers tasks can be grouped in classes, where each class represents tasks that require the same set of global resources with identical pattern of access operations to each of them. In *n*-th point in time the task class definition:

$$C^n_k = \{t_i: IRW_i = IRW_k \wedge IR_i = IR_k\} \quad (1)$$

where:

n – represents value of logical time,
$\wedge$ – represents logical conjunction,
IRW_i – bits sequence that represents resources read or write by task t_i,
IR_i – bits sequence that represents resources only read by task t_i

Each task class has own FIFO queue, which determines execution order for waiting tasks from the same class. Requested task is allocated to queue by its binary resource identifiers *IRW* and *IR*, which represents group of global resources that are required by task with its access pattern (read only/write only/read and write operations). Class task queues are emptied according to conflictless schedule determined to *n*-th point in time. Each task has timestamp determined by logical time, which represents moment of its appearance in task processing environment. Therefore conflictless scheduling bases on correct managing of class queues, where all tasks executed in parallel have no resource conflict. Each finish of active task and change in class task queues determine next point in logical time. This also means that all task executions with distinction to start and finish, have to be reported (even task execution finish with error). This allows to control, which waiting tasks can be executed, when last active conflicted task finish its execution. Not knowing the finish time of active tasks we can wait for completion one of them and then calculate suitable conflictless schedule, but this introduces additional delays in task processing environments. Alternatively we can calculate in advance many conflictless schedules, one for each active task. This approach requires efficient processing environment for preparation many conflict less schedules, each one S^n_k for every active task $t_k \in R^n$ and for fixed *n*-th point in time. To calculate schedules effectively was chosen massively parallel processing environment supported by modern GPU card. However development of dedicated GPU algorithm to effective calculation of conflictless schedules requires fast resource conflict detection between tasks. This problem was solved using two binary resource identifiers IRW and IR, which are assigned to each task and represents global resources that are used by this task operation with distinction read only resources.

Binary resource identifiers allow rapid conflict detection between tasks, details and examples are presented in other publications [4, 5]. The resource conflict detection between two tasks belonging respectively to class C^n_x and C^n_y can be check by simple condition:

$$(IRW_x \; and \; IRW_y) \; xor \; (IR_x \; and \; IR_y) \neq 0 \tag{2}$$

If above condition is satisfied, then exists at least one resource conflict between task classes, determined by only three simple binary operators and one comparison. This means that every two tasks, from those classes cannot be

executed in parallel, its execution order has to be determined by its logical time values. If resource conflict is detected between tasks, then longer waiting task has to be executed at first. This rule prevents tasks starvation and deadlock because some task could get stuck in the class queue [5].

3. The GPU algorithm for determination conflictless schedule

Modern GPU processing environment is an example of SIMD (Single Instruction stream Multiple Data streams) in Flynn's taxonomy in contrast to task processing environments that are classified as SISD (Single Instruction stream Single Data stream) [2]. Single GPU card is isolated processing environment because processing units can share GPU memory area (i.e. global memory), but cannot directly access to memory located beyond GPU card. Designing of GPU algorithm requires taking into account all memory transfers between host system and GPU memory. To avoid unnecessary data transfers and repeated calculations we propose some dedicated data structures: conflict array and conflict lists. A conflict array stores information about resource conflicts between classes. Whereas a conflict list is kept for each task class and stores all classes that are in conflict with it. Task classes are identified on list by numbers and the conflict list contains only selection of those conflicted classes, that queues are not empty. The order of task classes in conflict list is fixed by timestamps of oldest task waiting in class queue.

Developed algorithm has four phases - respectively denoted: SORT, CAT, ACS, CLS, that are processed sequentially. Each phase has prepared computation kernel dedicated to GPU processing. To determine a conflictless schedule, all phases have to be processed sequentially, because the calculation results of the previous stage are used in next computation phase. It should be noted that those results between computation stages are stored in GPU memory, which minimize unnecessary data transfer between host and GPU memory. Additionally due to the features of the GPU memory (in particular of the restrictions on the local memory), in each phase of presented algorithm, all computations have to be divided into work-groups. All calculations in a single work-group are run in the same computation unit, using its many processing elements. In each phase has to be established a number of work-groups and its dimensions. In all phases of presented algorithm all work groups have only one dimension, but its size varies.

In first computing SORT phase all task classes have to be ordered by timestamps of oldest waiting task located in class queue. This first stage of presented algorithm provides classes sorting in descending order determined by logical time values. To solve this problem in massively parallel environment there was used sorting by counting, because assigned to tasks logical time values are unique [1]. Each GPU processing unit has to calculate positions in sorting order for

fixed number of classes, which is determined by work-group size. Calculated positions are stored in local memory; this allows to significantly reduce number of write operations in global GPU memory. Due to limitation of local memory size the work group size was statically fixed to 16. Number of work-groups in first phase is calculated as number of all classes with not empty queues divided by number of work group size. Therefore in processing unit each one with 16 processing elements has to count for fixed class number, how many classes have greater value of logical time for oldest waiting task. All calculated counter values are stored in local memory and sets class positions in sorting order. At the end of the first phase numbers of sorted classes are stored in global memory, they will be used in next computation phase of presented algorithm.

Second computing CAT stage is responsible for calculations of conflict class array and lists of conflicted classes. Similarly, the number and size of work-groups are identical as in previous phase. Resource conflicts between classes is determined according to the relation (2), calculations are performed for all pairs of task classes. Each processing element for fixed class verifies conflict with any other task class, recording results in conflict array as boolean value and store all numbers of conflicted class in conflict list. This conflict list is created individually for each task class. Class order in resource conflict verification is fixed according to previous computation phase. In example, for fixed class the conflicted class with the oldest waiting task will have always first position on its list of conflicted classes. It also should be noted that each class on conflict list is a candidate to participate its conflictless schedule, when active task from fixed class finishes its execution.

In third ACS phase a set of active classes is selected. This selection shows classes that tasks are executed in parallel without conflicts in access to global resources. Knowledge of active classes is required to establish the number of conflictless schedules that will be calculated in next phase of presented algorithm. Calculations of largest collection of active class are performed in single work-group, which has maximum size determined by GPU specification (usually work-group size is limited up to 1024). In this phase the data from conflict array and conflict list are used, which improves computation.

The last computation CLS phase of proposed algorithm is responsible for determination of conflictless schedules, which number is determined by cardinality of active class set R^n. This fourth algorithm phase is the most important for conflictless task scheduling, because for each active task it calculates the set of classes which tasks can start execution in parallel immediately after this active task finish. In CLS phase number of work-groups is equal with cardinality of active class set, so every processing unit calculates at once only one conflictless schedule for single active class. Size of work groups is determined dynamically according to the maximum number of classes in conflict lists, prepared for the active classes.

Calculation of single conflictless schedule for one of active classes bases on content its conflict list. According to list order among classes has to be selected this subset, that have no resource conflict each other. Calculations of this subset is realised by many processing elements that divide this work between each other. This allows for efficient use of GPU resources to perform all operations in this algorithm phase. In CLS computation phase was used optimisations in access to memory areas. In each iteration of CLS phase class candidates to conflictless schedule were temporary stored in local memory of processing units. After then a subset of class candidates was written in global memory, so in each iteration number of writes to global memory was minimised.

Presented phase decomposition of conflictless scheduling algorithm show how features of massively processing GPU environment are adjusted to efficient determination of conflictless schedule. How much presented deterministic conflictless scheduling algorithm is dedicated to modern GPU, it will be shown in next chapter. This algorithm was implemented using OpenCL standard in software simulator of task processing environment, which using modern GPU card can calculate conflictless schedule adapted to state of task processing environment with high contention of global resources [3]. Prepared software simulator allows for define various task classes environment and for each of them can perform all phases of presented algorithm. Additionally simulator software measures time of each phase computation using GPU and CPU, which allows comparing performances in calculations of conflictless schedule in both computing sources. The measured results of computation, for example scenario of task processing environment, are presented in next section.

4. The calculation performance of conflictless schedules

The created simulator software will be used to present results of computation adaptive conflictless schedule in various task processing environments. This simulator implements conflictless scheduling algorithm presented in previous chapter. All calculation results were made in GNU/Linux operating system from CentOS 7.2 distribution with kernel version 3.10.0-327.36.3.el7.x86_64 using two computing sources: GPU and CPU. The computer specification includes Intel Core i5-2400 with 32 GB of RAM. The GPU used in computation is NVidia GTX 980 Titan with 2816 processing elements and 6 GB of GDDR5 memory with OpenCL version 1.2 and drivers from CUDA software in version 7.5.30. The GPU specification determines limits of task classes and global resources in environments. In the presented configuration the number of classes is up to 16368 and number of global resources is limited to 32768.

The computation performance of adaptive conflictless schedule will be presented for sample task processing environments with high contention of resources. Each testing scenario includes other task processing environments, which has fixed number of task classes and their conflict dependencies, also number of global resources that are used by tasks is fixed for each scenario. Performance results presented for each testing scenario will include total time of calculation conflictless schedule using only CPU and GPU. Separately computation times are presented for each phase of conflictless scheduling algorithm, additionally times of data transfers between GPU and host computer memory are shown in each algorithm phase. This allows to reliable compare the performance calculations of conflictless schedules using massively parallel GPU processing and sequential processing with single core of CPU.

The simulator software create all data for selected scenario defined by number of task classes and number of global resources, it assumes that in all tasks classes are waiting tasks and there are resource conflicts between them. In all scenarios each task belongs to only one class and has to use many global resources that generate resource conflicts between them. All binary resource identifiers for classes and timestamps for oldest tasks located in class queues are prepared by simulator software.

The first example of scenario has 5000 task classes and 64 global resources. The level of conflicts between those classes is 4.6%, number of classes that tasks are active is 32. Therefore in this scenario 32 conflictless schedules will be calculated using CPU and GPU. The computation performance for first scenario with timing results of all phases of conflictless scheduling algorithm is presented in Table 1.

Table 2. Performance results for conflictless scheduling calculated with CPU and GPU for scenario with 5000 task classes and 64 resources

	GPU transfer time (µs)	GPU processing time (µs)	GPU transfer and processing time (µs)	CPU processing time (µs)
SORT	199	32	231	24647
CAT	458	421	879	80051
ACS	10	38	48	209
CLS	902	16	918	108
total	1569	507	2076	105015

Results from Table 1 show that in total GPU processing of 32 conflictless schedules is over 200 times faster than processing with CPU. In all algorithm phases this data processing time for GPU is better than using one core CPU. Including data transfer to and from GPU memory calculations of conflictless schedules are in total 50 times faster than using CPU.

Table 2. Performance results for conflictless scheduling calculated with CPU and GPU for scenario with 16368 task classes and 64 resources

	GPU transfer time (µs)	GPU processing time (µs)	GPU transfer and processing time (µs)	CPU processing time (µs)
SORT	241	146	387	755985
CAT	2623	2174	4797	112396
ACS	11	40	51	694
CLS	1510	17	1527	577
total	4385	2377	6762	869652

Second task processing environment has 16368 task classes and 64 global resources. Also in second scenario level of conflict between classes and number of active classes is the same as in first scenario. The performance results in calculation of conflictless schedules for second scenario are presented in Table 2. Comparison of CPU and GPU processing time in second scenario shows that in total GPU is 365 times faster than CPU. Including GPU data transfers its performance dominance is 128 times faster. This scenario demonstrate also that increasing number of task classes causes observable rise time in SORT and CAT phases of conflictless scheduling algorithm, because there are dependent on the number of task classes. Processing in ACS and CLS phase of algorithm is dependent on number of active classes.

The third scenario represents task processing environment that has 16368 task classes and 1024 global resources. Number of active classes is 202 and the level of conflicts between task classes is 1.2%. The computation performance for third scenario with timing results of all phases of conflictless scheduling algorithm is presented in Table 3.

Table 3. Performance results for conflictless scheduling calculated with CPU and GPU for scenario with 16368 task classes and 1024 resources

	GPU transfer time (µs)	GPU processing time (µs)	GPU transfer and processing time (µs)	CPU processing time (µs)
SORT	216	154	434	751790
CAT	3661	2086	5747	13463
ACS	12	40	52	4670
CLS	5310	175	5485	3230
total	9263	2455	11718	773153

In third example scenario GPU processing was 314 times faster than CPU and including data transfers it was 66 times faster. In comparison to previous scenarios number of global resources is 16 times greater, what caused respectively longer resource binary identifiers. This especially affects duration of CAT and CLS phases of algorithm. Generally greater number of global resources extends calculations of

conflictless schedules. For CLS phase is also important number of active classes those tasks are active. This determines number of conflictless schedules to process in CLS phase.

As observed in examples scenarios data processing in all algorithm phases is realized more efficiently using GPU. This ratio is not always profitable in case, when also GPU data transfers are included. For example in CLS phase processing and data transfers by GPU is not efficient as CPU. The reason is transferring from GPU in CLS phase all calculated conflictless schedules, from which only one will be used. The software simulator transfers them only to print all determined conflictless schedules.

5. Conclusions

The created simulator software verifies the task coordination concept using modern GPU card and dedicated conflictless scheduling algorithm. It also enables the performance measurement of GPU processing in determining the conflictless schedules for various task processing environments with high contention of resources. Adjustment of conflictless scheduling algorithm to massively parallel processing by GPU and division to calculation stages assures the effective calculation of conflictless schedules. As show by the results of experiments using modern GPU conflictless schedules can be calculated in milliseconds using deterministic algorithm. If time resolution of task executions also is in milliseconds then calculations of conflictless schedules can be performed in advance. Then conflictless schedule can be determined before execution of active task finishes. This is because GPU card provides isolated computing environment, which resources are not used in task processing environments.

The GPU efficiency in determination of conflictless schedules was confirmed in experiments, where computation results were obtained by GPU many times faster than by single core of CPU. Even including GPU data transfers the results of computation were better that using CPU. This allows to conclude, that novel approach of task coordination in access to global resources can be done efficiently using GPU conflictless scheduling. This approach can significantly facilitate developments of multitasking software in environments with high contention of resources. Software developer using conflictless scheduling is not obligated to use any other synchronization mechanism, because all resources conflicts will be eliminated automatically. There is also no possibility to the occurrence of task starvation in access to the global resources, because conflictless scheduling guarantees them access in a finite time. Even deadlock between tasks cannot occur due to wrong global resources allocation.

The conflictless scheduling is designed to high contention environments with limited number of global resources, where set of all required resources is known

for each task before its execution begins. As established in computation experiments also task execution time should be longer that minimum limit to determine required conflictless schedule in advance. This minimum task duration limit is dependent on task environment parameters like number of global resources and number of task classes, also it is dependent on GPU specification. The GPU card equipped with more memory or more efficient processing elements will provide faster determination of adaptive conflictless schedule than presented in this paper.

REFERENCES

[1] Amato N., Ravishankar Iyer R., Sundaresan S., and Wu. Y. (1998) *A Comparison of Parallel Sorting Algorithms on Different Architectures*. Technical Report. Texas A & M University, College Station, TX, USA.

[2] Flynn M.J., Rudd R. W. (1996) *Parallel architectures*, ACM Computing Surveys, Volume 28, Issue 1, 67–70

[3] Martineau M., McIntosh-Smith S., Boulton M., Gaudin W. (2016). *An Evaluation of Emerging Many-Core Parallel Programming Models*. In Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'16), Pavan Balaji and Kai-Cheung Leung (Eds.). ACM, New York, NY, USA

[4] Smoliński, M. (2016) *Coordination of Parallel Tasks in Access to Resource Groups by Adaptive Conflictless Scheduling*. Beyond Databases, Architectures and Structures. Advanced Technologies for Data Mining and Knowledge Discovery

[5] Smoliński, M. (2016) *Elimination of task starvation in conflictless scheduling concept*. Information Systems in Management Vol. 5, No. 2, 237–247

[6] Stallings W. (2015) *Operating systems, Internals and Design Principles*. Pearson Education, 8th edition

[7] Tanenbaum, A., Bos H. (2014) *Modern operating systems*. Prentice Hall, 4th edition