공학석사학위논문

Slice Counts Search for Real-Time Guarantee and Better Schedulability of GPU

GPU의 실시간 보장 및 더 나은 스케줄링 가능성을 위한 슬라이스 수 탐색

2021년 8월

서울대학교 대학원
컴퓨터공학부
박 하 연

Slice Counts Search for Real-Time Guarantee and Better Schedulability of GPU

GPU의 실시간 보장 및 더 나은 스케줄링 가능성을 위한 슬라이스 수 검색

지도교수 이 창 건

이 논문을 공학석사 학위논문으로 제출함

2021년 5월

서울대학교 대학원

컴퓨터공학부

박 하 연

박하연의 공학석사 학위논문을 인준함

2021년 6월

Abstract

Slice Counts Search for Real-Time Guarantee and Better Schedulability of GPU

Hayeon Park
Department of Computer Science and Engineering
The Graduate School
Seoul National University

This paper proposes a conditionally optimal slice counts searching algorithm to improve GPU's real-time guarantee and better schedulability. Despite the growing importance of GPUs due to the recent advances in deep learning, there is still a lack of technology to utilize them in real-time. This paper assumes a GPU as a uniprocessor and uses non-preemptive EDF to schedule GPU kernels. Additionally, solving the schedulability degradation problem caused by non-preemptive uniprocessor assumption through searching the slice count of each kernel that makes the GPU task set to be schedulable.

keywords : Real-Time Systems, GPU, Task Slicing

Student Number : 2019-28371

Contents

List of Figures

1 Introduction

Emerging cyber-physical systems such as autonomous driving vehicles require massive computations for object classification, localization, and deep learning, etc. Furthermore, those computations need to be finished before tight deadlines. Thus, it is inevitable to use GPUs that has massive internal parallelism with thousands of internal cores. However, once a computational unit called a GPU kernel starts executing on the GPU, all other GPU kernels should wait until the current GPU kernel completes. That is, the GPU is a non-preemptive resource from the GPU users' point of view. Therefore, if there are multiple concurrent real-time tasks that uses the GPU, the overall schedulability is significantly reduced because of the non-preemptivity of the GPU.

In order to address this issue, the lastly released NVIDIA GPUs with Pascal or more advanced architecture provide preemption functions. However, they are not visible from the programmer's point of view and hence hard to be used for the real-time guarantee. Also, those GPUs with the preemption functions are employed only in expensive high-end NVIDIA boards such as xxx. Most of cost-effective GPUs that are popular in the market do not provide preemption functions. Targeting such cost-effective non-preemptive GPUs, recent works [1–4] propose task slicing mechanisms that slice a long GPU kernel into short slices so that the non-preemptive duration of the GPU can be reduced to improve the real-time schedulability. Such slicing mechanisms can be realized outside of the GPU without changing GPU internals such as GPU internal scheduling and GPU drivers. Thus, they are attractive approaches that can be practically used for a wide spectrum of various GPUs whose internals are mostly hidden. However, it is not well studied how to optimally slice GPU kernels for guaranteeing the schedulability of multiple concurrent real-time tasks.

This paper proposes a conditionally optimal slicing algorithm that optimally decides slice counts for GPU kernels issued from multiple real-time CPU tasks. For this, we model the GPU as an non-preemptive uniprocessor and GPU kernels issued from CPU tasks as a set of sporadic GPU tasks that are scheduled by the non-preemptive EDF on the uniprocessor. For such a GPU task set, we propose a polynomial(?) complexity algorithm that incrementally compares the time-demand and time-supply for the non-preemptive EDF scheduling of the given GPU task set to compute the tolerable priority inversion durations due to non-preemption until the deadlines. From the tolerable priority inversion durations, the proposed algorithm determines the maximum possible slice sizes of the GPU tasks under the constraints of the tolerable priority inversion durations. Finally, the proposed algorithm determines the minimum slice count of each GPU task to minimizes the slice overhead such that each slice becomes smaller than the maximum possible size. Although, our slicing algorithm is just for the schedulability of the GPU tasks, we explain how it is used for improving the overall schedulability of the entire system in practical cases where each real-time task consists of a interleaved sequence of CPU segments and GPU segments.

In addition, we formally prove that our slicing algorithm is optimal for the schedulability under the conditions that (1) the slices are scheduled on a uniprocessor by the non-preemptive EDF, (2) the slicing overhead is proportional to the slice count, (3) a GPU kernel is evenly partitioned into slices. Our extensive experiments by both simulation and actual implementation shows that our optimal slicing can improve the schedulability upto 73%.

The rest of this paper is organized as follows. Section II presents related works. Section III shows the assumption for guaranteeing real-time of all type of GPU, Section IV formally defines our problem of task set slicing. Section V describes the feasibility test of non-preemptive EDF and how to find the optimal slice count of each task. Section VI reports our simulation experiment, and the conclusion of the paper is described in Section VII.

2 Related Works

A lot of research has been conducted to improve the schedulability of GPUs by identifying characteristics of resources and utilizing them well. TimeGraph[5] proposed a method for priority-based scheduling by identifying the trade-off between throughput and response time to improve the scheduling capability of graphical GPU tasks. ElasticKernel[6] was proposed to solve the fact that all resources are not used when GPUs are running, and research on scheduling performance improvement through partitioning of Streaming Multiprocessors(SM) in GPUs was also done[7]. References [8, 9] raised a problem with basic GPU scheduling and introduced a method to help select the appropriate GPU task during scheduling by tracking GPU usage at the OS or hypervisor level.

Memory contention is one of the major causes of GPU performance degradation. To solve this problem, Gdev[10] virtualized the GPU runtime to run inside the OS so that multiple GPU contexts share GPU memory and GPUs execute multiple logics. GPUfs[11] extends the area that developers can optimize by enabling GPU file I/O by extending the CPU buffer cache to the GPU memory area. GDM[12] allows different applications to share GPU resources by performing GPU memory management at the OS level. Sigamma[13] suggested a method to minimize memory contention in an integrated GPU (iGPU) where the GPU and CPU share system memory. However, methods of improving the efficiency of GPU resource usage cannot be any solution to the problem of deterministic task blocking, and there are limitations in achieving the purpose of a real-time guarantee of GPUs.

Various studies related to activation of preemption in GPU have also been conducted to solve such a problem. Reference [14] introduced a method to make data transmission preemptible by splitting a memory copy. However, there is a constraint that GPU code execution excluding data transmission is still non-preemptible. The SM in the GPU is scheduled in block units, and in one SM, the same instruction is executed for different data used by the scheduled block. Considering these characteristics, various methods to activate preemption by slicing GPU tasks to the block group level have been introduced[1–4]. These methods improve scheduling performance by slicing tasks with specified granularity and making the GPU itself non-preemptive in reality but making the tasks operate as if they were preemptive. These solutions have an advantage in that they have limited preemption, but because the preemption point is limited, it is difficult to guarantee that the target task set is always schedulable. As a result, it is impossible to directly use the various studies related to preemptive uniprocessor[15–20]. Also, these methods have a weakness in that they have slice overhead. To minimize this overhead, various methods to solve through hardware extension have been studied.[21–23] Although these efforts, real-time is still not guaranteed on GPUs because they do not slice task sets witn considering real-time constraints.

For NVIDIA GPUs, Pascal and later architectures provide pixel/thread level preemption. Based on this function, research has been conducted to minimize slicing overhead and enable single block-level preemption at any time.[4] Also, by utilizing the preemption of NVIDIA GPU, Earliest Deadline First(EDF) scheduling is performed for GPU tasks, and for misbehaving tasks, a Constant Bandwidth Server (CBS) is used to enable real-time scheduling of GPUs.[24] However, these methods are difficult to apply practically because these methods cannot be used for all types of GPU which pixel/thread level preemption mechanism is not supported.

Motivated by these limitations, in this paper, we will introduce a task set slicing algorithm to obtain maximum real-time schedulability by slicing a task set applied to all types of GPUs.

3 Real-Time Gaurantee of GPUs through Non-Preemptive Uniprocessor Assumption

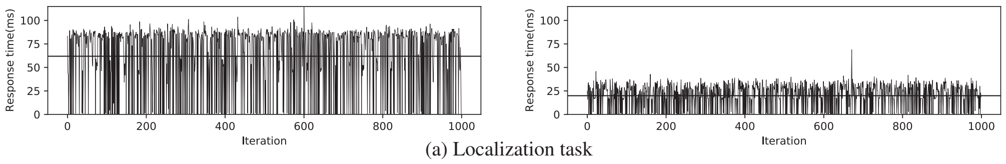
This section introduces the fact that, when assuming a GPU as a non-preemptive uniprocessor, it is relatively easy to guarantee real-time performance of the GPU in all types of GPUs, and in some cases, it is more effective compared to the case where it is not applied.

The fact that it is difficult to take advantage of preemption on GPUs is one of the biggest obstacles to real-time guarantees on GPUs. Since GPU is basically focused on maximizing throughput, predictability is very low. To solve this problem, it is necessary to use a real-time scheduling algorithm for GPU kernels. In general, a preemption-based scheduler is often used when constructing a real-time system in a CPU environment. Still, it is very difficult to utilize the preemption function because the GPU does not provide a preemption function or is very limited and not visible at the user-level. There is a case of implementing deadline-based scheduling that can be preempted for a specific GPU, but this has a limitation applicable to only a few Nvidia GPUs that provide a hypervisor function.

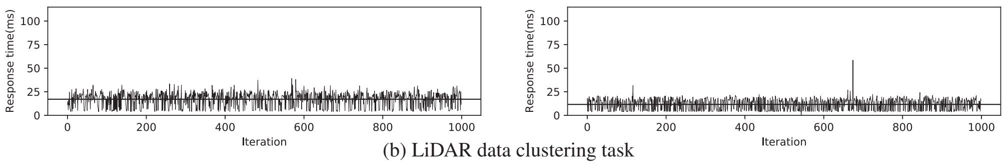
The second hindrance is that what is known about the internal scheduling policy of the GPU is limited, and there is no way to manipulate it. The theoretical operation method of the GPU is publicly disclosed, but the details of the implementation level are not known. This fact causes the user to be unable to predict the exact operation of the GPU. This problem is noticeable when performing concurrent kernel execution. Although the GPU itself can execute multiple kernels at the same time, it is not easy to predict exactly in what order the input kernels will be executed. Also, since it is also impossible to assign each kernel to a specific SM, it is also impossible to apply global scheduling assuming the SMs in the GPU as one processor.

These problems can be easily solved by assuming that the GPU is a non-preemptive uniprocessor. Since the GPU operates with its own non-preemptive policy, it can be seen that the non-preemptive assumption is valid. In addition, from the viewpoint of implementing the scheduler by itself, the non-preemptive assumption is relatively simple to implement compared to the preemption-based scheduling, and the scheduling overhead is small. It has great advantages in terms of assumptions. Also, in the case of the uniprocessor assumption, it is a realistic option that can be applied in reality. Crucially, this assumption has meaning in that it can be applied to all types of GPUs.

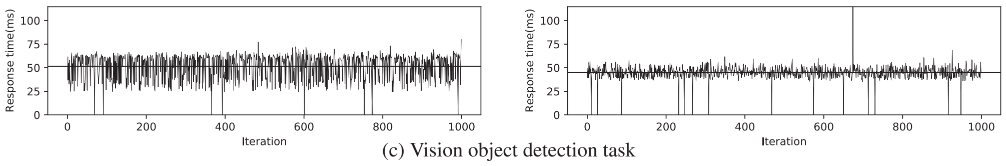
These problems can be solved by assuming that a GPU is a non-preemptive uniprocessor. Because GPU itself operates with a non-preemptive policy, assuming



(a) Localization task



(b) LiDAR data clustering task

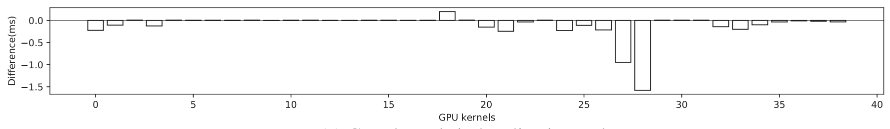


(c) Vision object detection task

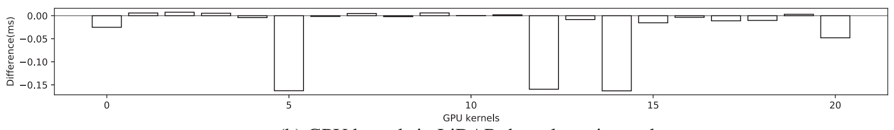
Figure 1: Response time of GPU tasks

it as a non-preemptive processor is valid. Additionally, non-preemptive assumption makes it relatively simple to implement scheduler than preemptive based scheduling. Also, the uniprocessor assumption is the realistic option to build a real-time system of a GPU. Crucially, these assumptions are meaningful in that they can be adapted to all types of GPUs.

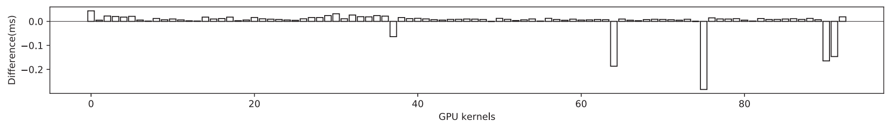
The increase of response time caused by assumptions can be seen as indispensable to gain predictability, and rather it decreases in specific cases. Figure. 1 shows the response time of GPU tasks used for autonomous driving in Autoware.AI when all tasks are executed simultaneously. The whole profiling process was conducted on the Nvidia Xavier with Ubuntu 18.04. The left side graphs are the case use the default Ubuntu scheduler. The right side is the case of adapting non-preemptive assumptions and use EDF scheduling. The horizontal line of each graph represents the average response time of each process. The software level non-preemptive uniprocessor EDF



(a) GPU kernels in localization task



(b) GPU kernels in LiDAR data clustering task



(c) GPU kernels in vision object detection task

Figure 2: Difference of average response time of GPU kernels

scheduler is used for profiling.

As can see in the graph, the average response time always becomes smaller when using non-preemptive uniprocessor assumption. The reason why response time reduction has occurred is that the reduction of memory contention. Since the L2 cache is shared by all SMs, the concurrent kernel execution causes more memory contention. However, if GPU operates as a non-preemptive uniprocessor, there is a relatively small chance to cause L2 cache miss, although the L2 cache size of Nvidia Xavier is only 512KB. Therefore, in the case of the computation unit of GPU is not enough to execute many tasks simultaneously, the assumptions can reduce the response time of the task. The one more important difference between cases is that the response time variance at the non-preemptive uniprocessor case is much smaller than the default case. This phenomenon increases the predictability of tasks, and it makes the system more reliable. However, some peak points are created at assumption case because the response time can be greatly increased when a lot of GPU kernels exist in the task and priority inversion is occurred frequently.

Figure. 2 shows the difference in average response time of each GPU kernel in tasks in the same environment. The x-axis represents the specific id of each GPU kernel, and the y-axis represents the difference calculated by subtracting the default case from the non-preemptive uniprocessor case. Since priority inversion occurs by non-preemption and only one kernel can be executed simultaneously, the average response time of most kernels is slightly increased. Still, the average response time is decreased dramatically in some kernels. Because the decreasing amount is relatively much huge than increments, the total response time of the task is reduced. This phenomenon shows that the overhead of the L2 cache is significantly huge in the GPU, and it can be reduced by non-preemptive uniprocessor assumption.

However, as seen in the peak points in the non-preemptive uniprocessor case, it can produce deadline miss that cannot be avoidable because the non-preemptive duration is increased than before. This problem can be solved by slicing the GPU kernels. It can be conducted in many different ways, i.e., block-level slicing or divide the GPU operation into some pieces. The best way to find the slice count of each GPU kernel is introduced in the rest of the paper.

4 Problem Description

In this paper, we consider a system with a multiple CPUs and a single GPU that run a set Γ of n sporadic tasks.

$$\Gamma = (\tau_1, \tau_2, \cdots, \tau_n)$$

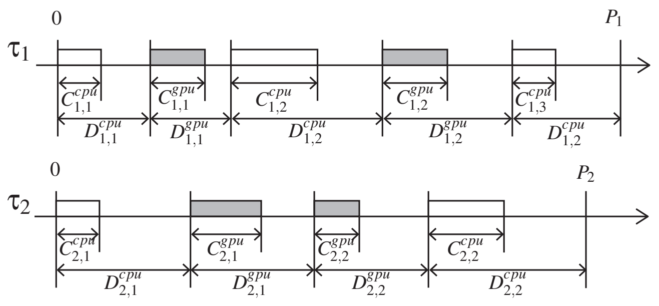


Figure 3: CPU/GPU Mixture Task Model

Each task τ_i is represented as a 3-tuple $\tau_i = (P_i, C_i, D_i)$, P_i is the minimum inter-release time, C_i is the worst case execution time, and $D_i (D_i \leq P_i)$ is the relative deadline. Typically, each CPU segment in a task that uses GPUs is interleaved by single or multiple GPU segments that corresponding to GPU kernels in CUDA. Therefore, they can be represented as a CPU/GPU mixture model like Fig. 3.

A k-th CPU segment is represented by $\tau^{cpu}_{i,k} = (P_i, C^{cpu}_{i,k}, D^{cpu}_{i,k})$ and a k-th GPU segment is represented by $\tau^{gpu}_{i,k} = (P_i, C^{gpu}_{i,k}, D^{gpu}_{i,k})$. Because they are included in τ_i, minimum inter-release time of each segment is same with P_i. All segments except last one are precedence constraints for each other and it makes segments in the same task cannot be executed simultaneously.

Problem Description: For all GPU segments in Γ, our problem is to find the $\mathbb{SC}$, set of slice counts for each GPU segment, that makes all GPU segments are scheduled by non-preemptive EDF on the GPU with meeting their deadline.

A slice count for a GPU segment $\tau^{gpu}_{i,k}$ is represented by $sc_{i,k}$ and the $\mathbb{SC}$ consists of slice counts of all GPU segments. When a segment $\tau^{gpu}_{i,k}$ is sliced to m pieces, a overhead due to the slicing is denoted by $O_{i,k}(m)$ and assume that the original computation amount $C^{gpu}_{i,k}$ and the slicing overhead $O_{i,k}(m)$ together is evenly partitioned

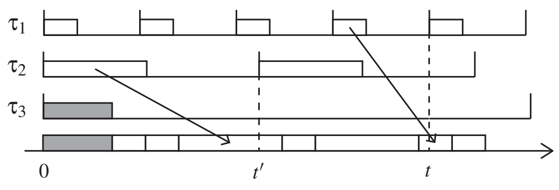


Figure 4: Example of deadline miss due to the priority inversion

into m slices. Therefore, each slice size becomes

$$\frac{C^{gpu}_{i,k} + O_{i,k}(m)}{m}$$

and it is the maximum non-preemptive duration may cause the priority inversion to other GPU segments with earlier deadline.

At first, we present how to search slice counts for the single task set and after that address how it can be used for determining the $\mathbb{SC}$ for all GPU segments in Γ where each task τ_i consists of an interleaved sequence of multiple CPU/GPU segments.

5 Slice Counts Search

This section denotes a task only as τ_i to focus on slice counts search for single task set Γ without any other representation like $\tau^{cpu}_{i,k}$ and $\tau^{gpu}_{i,k}$. Also, the slice overhead of τ_i when sliced to m pieces is denoted by $\O_i(m)$.

The proposed slice counts search is motivated by the exact analysis method for EDF on non-preemptive uniprocessor[25]. It uses discrete time units for clarity, but in this paper, we consider continuous time for practicality. When L is the synchronous busy period, the condition for the feasibility test is as follows.

For any general task set with $U \leq 1$ is feasible, using EDF, if and only if

$$\forall t \in S, h(t) \leq t$$

when

$$h(t) = \max_{D_j > t}\{C_j\} + \sum_{D_i \leq t}\left(1 + \left\lfloor \frac{t - D_i}{P_i} \right\rfloor\right) \cdot C_i \tag{1}$$

$$S = \{\cup_{i=1}^{n}(k \cdot P_i + D_i), k \in \mathbb{N}\} \cap [0, L) \tag{2}$$

The $h(t)$ represents a maximum processor demand at t. Unlike a processor demand for the preemptive EDF, it has an extra load $\max_{D_j>t}\{C_j\}$ for a processor demand due to the priority inversion at 0. In the preemptive case, a maximum processor demand at t is generated when all tasks are simultaneously released at 0 and all task τ_js whose $D_j > t$ are not involved to it. But in the non-preemptive case, when τ_j is released before the t and its absolute deadline is smaller than t, it creates non-preemptive duration and others are blocked until it is finished. Therefore, maximum extra load is generated when τ_j whose C_j is biggest among tasks whose relative deadline is bigger than t is scheduled just before the 0. If there is no τ_i that $D_i > t$, the extra load becomes 0.

Because a processor demand at t only includes an execution of a job whose absolute deadline is smaller than t, it looks like deadline miss due to the priority inversion is not considered when blocking task's absolute deadline is bigger than t. But when deadline miss occurs at t although its demand is smaller than or equal to t by this reason, there is no problem because it is always captured at some preceding time point t'. In Fig. 4, deadline miss is occurred but it meet the condition. But in the same

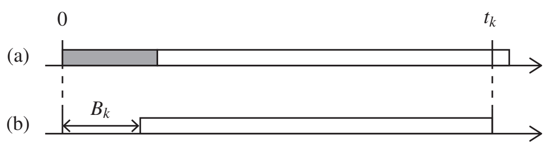


Figure 5: Blocking tolerance

scenario, deadline miss is captured at t', so this task set is determined as not feasible. Because all deadline miss scenarios can be converted to the worst case like Fig. 4, a task set is always schedulable when it always meets the condition for all $t \in S$.

5.1 Blocking point, blocking tolerance, and blocking candidates

Before searching a slice count of each task, it is necessary to know that since a task slicing only makes a non-preemptive task set to be pseudo-preemptive, its theoretical maximum schedulability is the same with a preemptive case. A slicing improves schedulability by reducing the length of blocking duration which creates an extra load. Therefore, if a deadline miss occurs when extra load is 0, a task set cannot be schedulable by slicing. An extra load is 0 when $t \geq \max_{1 \leq i \leq n}\{D_i\}$ and this fact allows us to limit the time point which we need to consider during searching. In this paper, we define the time point which an extra load exists as a blocking point and denote each blocking point by t_k. A t_k is always smaller than t_{k+1}.

The blocking tolerance B_k is the maximum value of a blocking duration causes an extra load at blocking point t_k with meeting the feasibility test condition. In Fig. 5(a), the grey bar represents the extra load and the white bar represents the processor demand from all τ_i that $D_i \leq t_k$. When excepts the extra load at $h(t_k)$ like Fig. 5(b), the rest duration of $h(t)$ can be the blocking tolerance. Therefore, the B_k can be calculated as follows.

$$B_k = t_k - \sum_{D_i \le t_k} \left(1 + \left\lfloor \frac{t_k - D_i}{P_i} \right\rfloor\right) \cdot C_i \tag{3}$$

All τ_js that $D_j > t_k$ can be the candidates of the extra load at t_k. We call them blocking candidates at t_k and denote them by Γ_k^{blk} in the rest of the paper. To meet the test condition for all $t \in S$, not only the task which causes the maximum extra load but all tasks in Γ_k^{blk} should be sliced to have a shorter execution time than B_k. Note that $\Gamma_{k+1}^{blk} \subset \Gamma_k^{blk}$ is always holds because $t_k < t_{k+1}$ and Γ_k^{blk} is the set of τ_js which $D_j > t_k$.

5.2 Searching slice counts for a task set

Slice counts search is performed to all blocking points in order. When the current blocking point is t_k, following four step process is conducted.

- **Step(1)**: Calculate the B_{min} which is the $\min_{1 \le i \le k}\{B_i\}$.
- **Step(2)**: Select target tasks to search slice count in current blocking point.
- **Step(3)**: Search the slice count of each target task with slicing overheads.
- **Step(4)**: Update blocking tolerances of all blocking points that bigger than t_k.

Because slicing overheads are used in **step(3)**, we assume that slicing overheads are given.

The B_{min} is the key parameter that determines the slice count of each task. Every task τ_i in Γ should have C_i smaller than or equal to the corresponding B_{min} to meet the test condition. It can be obtained by comparing B_{min} at the previous blocking point and current blocking tolerance.

When current blocking point is t_k, all τ_js included in $\Gamma_{k+1}^{blk} - \Gamma_k^{blk}$ cannot be the blocking candidate of every $t_l > t_k$. Unlike need to be smaller than or equal to all blocking tolerance at t_k or earlier blocking points, these tasks have no responsibility to be smaller than all B_ls. Note that there is no difference whether satisfying all blocking tolerance corresponding to before or at t_k, or satisfying only B_{min} when searching the slice count. Therefore, the $\Gamma_{k+1}^{blk} - \Gamma_k^{blk}$ can be the set of target tasks needed to be searching slice counts at t_k, and we can search slice counts of them by leveraging B_{min}.

The slicing count of τ_j at t_k can be easily obtained by using the following equation.

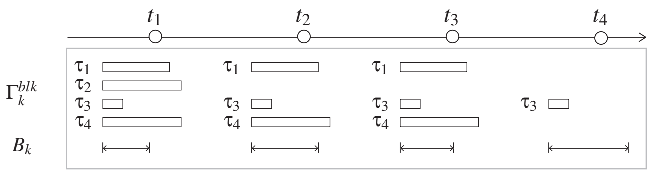
$$sc_j = \underset{m}{\operatorname{argmin}} \left\{ m \middle| \left\lceil \frac{C_j + O_j(m)}{m} \right\rceil < B_{min} \right\} \tag{4}$$

When the task τ_j is sliced at t_k, the second term of the $h(t_l)(t_l > t_k)$ is increased due to the slicing overhead of τ_j. This fact highlights the need that decreases the blocking tolerances of all t_ls. Note that τ_j cannot be the blocking candidate at t_l if its slice count is searched at t_k. The task which is not blocking candidates does not affect extra overheads. When τ_j is sliced to m pieces, the decrement value of the blocking tolerance B_l can be calculated as follows.

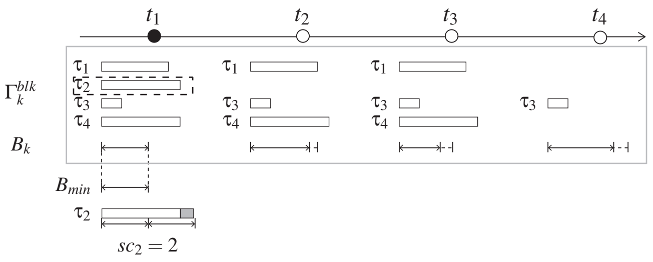
$$\left(1 + \left\lfloor \frac{t_l - D_j}{P_j} \right\rfloor \right) \cdot O_j(m) \tag{5}$$

This decrement value is calculated for every t_l and subtracted from all corresponding B_l. This process is called the blocking tolerance update.

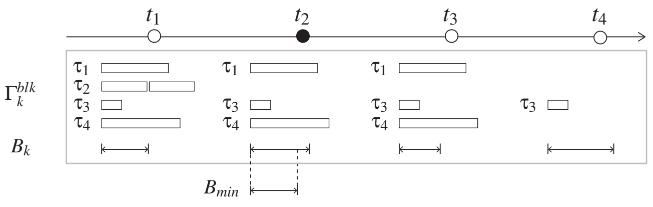
Fig. 6 shows the example of slice counts search that includes various scenarios.



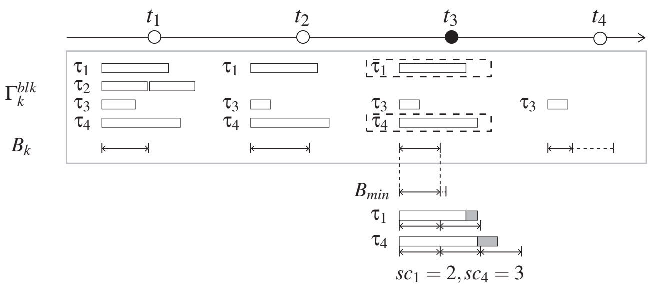
(a) Initial state



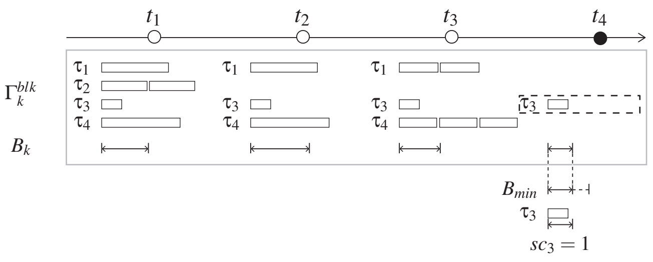
(b) First iteration



(c) Second iteration



(d) Third iteration



(e) Last iteration

Figure 6: The process of slice counts search

Initial state(Fig. 6 (a)): In this example, there is four blocking points. A circle below each t_k is colored to black when the slice count search is performed to the corresponding blocking point. Because this is the initial state now, no black circle exists. Each blocking point has blocking candidates Γ_k^{blk} and blocking tolerance B_k. The execution time of each task in Γ_k^{blk} is represented by a white bar. As mentioned earlier, $\Gamma_{k+1}^{blk} \subset \Gamma_k^{blk}$ is always holds.

First iteration(Fig. 6 (b)): At first, B_{min} is initialized by B_1 because no other blocking tolerance exists. The target tasks are wrapped in a dotted box. In this iteration, the target task is τ_2 because it is not included in Γ_2^{blk}. Below the B_{min}, slice count is searched. The dark grey bar next to the execution time of τ_2 represents the slicing overhead $O_2(2)$. Because sum of C_2 and $O_2(m)$ is within two pieces of B_{min}, sc_2 becomes 2. Because the slicing overhead is added, the blocking tolerances of t_2 to t_4 are decreased. The decrement is represented by the dotted line in the figure.

Second iteration(Fig. 6 (c)): Because previous B_{min} is shorter than B_2, B_{min} is not updated in this iteration. Furthermore, there is no target tasks because $\Gamma_2^{blk} = \Gamma_3^{blk}$. In this case, skip remain steps and move to next iteration.

Third iteration(Fig. 6 (d)): This iteration is almost same with first iteration. The B_{min} is updated to the B_3 and target tasks is set to the τ_1 and τ_4. There is no difference in slice count search when there is a single target task and multiple target tasks. The sc_1 and sc_4 are set to be 2 and 4, and B_4 is updated.

Last iteration(Fig. 6 (e)): The only difference between previous iteration and current iteration is that the target tasks is same with Γ_4^{blk} because there is no more blocking points. The sc_3 becomes 1 because C_3 is smaller than updated B_{min}.

5.3 Stop conditions

There are the two conditions that make it impossible to find the $\mathbb{SC}$ through the slice counts search. Because the theoretical maximum schedulability of slicing on non-preemptive EDF is the same as the case of preemptive EDF, the searching cannot find the $\mathbb{SC}$ when preemptive EDF cannot schedule the task set. The extra load of $h(t)$ in Eq. 1 becomes 0 when the time point $t \geq \max_{1 \leq i \leq n}\{D_i\}$. Therefore, if the test condition is missed at any point $t \geq \max_{1 \leq i \leq n}\{D_i\}$, there is no meaning to find the $\mathbb{SC}$. Also, since the basic rule of searching slice counts is finding the slice count which makes the size of the slice becomes within the corresponding blocking tolerance, it is also impossible to find the $\mathbb{SC}$ when any blocking tolerance B_k is smaller than 0.

5.4 Optimality of the slice counts search

The proposed slice counts search is conditinally optimal. The following theorem and proof show that.

Theorem 1. The proposed slice counts search is optimal in terms of the feasibility test for non-preemptive EDF on uniprocessor when the slicing overhead is proportional to the slice count and execution time of each slice is evenly divided from the original task. More specifically, if proposed approach cannot slice the task set that makes the given task set to pass the feasibility test, no other task set slicing algorithm do either.

Proof. The proof will show the optimality of the proposed approach by the following three steps:

- **Step 1**: proposed approach always finds the minimum slice count for every task if it is possible to make the task set feasible.

- **Step 2**: When $t < D_{max}$, if the proposed approach cannot make the feasible task set, no other task set slicing algorithm can do either.
- **Step 3**: When $t \geq D_{max}$, if the proposed approach cannot make the feasible task set, no other task set slicing algorithm can do either.

Proof of Step 1: When slicing is possible through the proposed approach, each task in the input task set Γ is always sliced to the minimum number. When $l > k$, target tasks for F_k are not always included in inversion candidates for all F_l. For this reason, when slicing the target task of F_k, the size of the first term of $h(t_l)$ is always increased, so the L_l value is decreased by Eq. 5. However, on the contrary, it does not affect the inversion tolerance value at a point earlier than t_k. The smaller the number of slices, the smaller the slice overhead, resulting in a smaller reduction in the inversion tolerance of the following points. proposed approach constantly searches the minimum slice count that the target task can pass through all associated inversion tolerances. This, in turn, makes the inversion tolerance that each task must pass as large as possible and allows all tasks to have the smallest slice count.

Proof of Step 2: Suppose that proposed approach cannot be made feasible through task set slicing for a set of tasks, but an arbitrary algorithm is possible when $t < D_{max}$. proposed approach always finds the smallest slice counts that make inversion duration less than inversion tolerance for all tasks. Therefore, if using a different algorithm, the slice count of each task is always greater than or equal to the case of the proposed approach. If the slice count increases, it makes it harder to find the slice count to pass the feasibility test increases because it reduces the size of the inversion tolerance that other tasks must pass. Because of this fact, the above assumption is erroneous, so if proposed approach cannot make a task set to be feasible, no other slicing algorithm can do either.

Proof of Step 3: Similar to what proof of Step 2, suppose that proposed approach cannot be made feasible through task set slicing for a set of tasks, but an arbitrary algorithm is possible when $t \geq D_{max}$. In this case, it can be proved simply by using the value of $h(t)$. When $t \geq D_{max}$ by Eq. 1, the value of $h(t)$ is determined only by the first term, and is proportional to the execution time of each task. proposed approach always finds the minimum slice count for all tasks, so at all times when $t \geq D_{max}$, it always has the smallest $h(t)$ value than when slicing task set with another combination of slice count. Therefore, if slicing is impossible with proposed approach, slicing is impossible with any other algorithm.

$\square$

5.5 Applying slice counts search in Real System

The tasks that include GPU kernels in the real system can be seen as a CPU/GPU multi-segment model like Figure. 7. Every CPU segment $\tau_{i,k}^{cpu}$ follows the task scheduler in CPU. All GPU segments $\tau_{j,l}^{gpu}$ follow the non-preemptive EDF as we assumes. The deadline of each segment can be determined by any prior study for deadline partition, i.e., prorating the task's deadline based on the WCET of each segment.

For guaranteeing real-time of GPU, we need to consider all possible set GPU segments. When the candidate of Γ^{gpu} is denoted as a Γ_j^{gpu}, all Γ_j^{gpu} can be generated by finding all combination that would occur if one GPU segment was selected for each task. Because multiple GPU segments in the same τ_i cannot be executed simul-

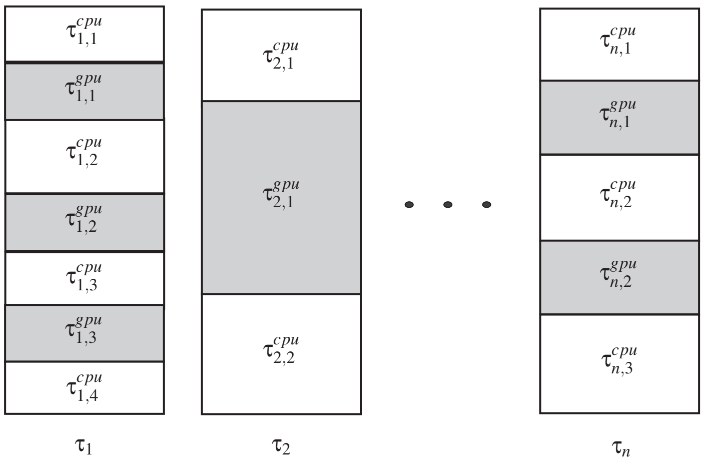


Figure 7: CPU/GPU multi-segments task set

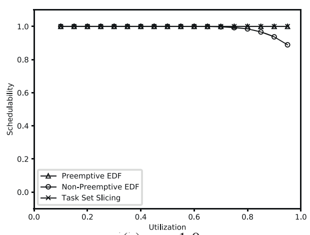
taneously, only one GPU segment per τ_i is included in Γ_j^{gpu}. Therefore, if n_i is the number of GPU segments in τ_i, total number of Γ_j^{gpu} becomes $n_1 n_2 \cdots n_n$. Because a slice count for a GPU segment can be different depending on the configuration of the GPU segment set, the final slice count for each GPU segment should be determined as a maximum value obtained while applying the slicing algorithm to all Γ_j^{gpu}.

6 Experiment Results

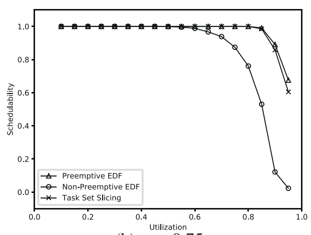
In this section, we show the effectiveness of the proposed algorithm by simulation with synthetic tasks and actual implementation with real autonomous driving tasks.

6.1 Simulation Experiment

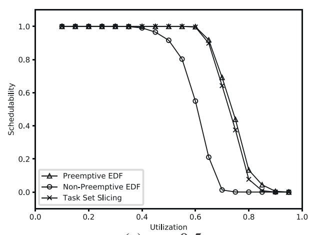
A synthetic task is used for whole simulation experiment. A task $T_i = (P_i, C_i, D_i)$ is randomly generated as follows: (1) P_i is randomly generated from uniform$[1000, 2000]$. (2) C_i is defined by multiplying the P_i and the task utilization. The task utilization is



(a) $\alpha = 1.0$



(b) $\alpha = 0.75$



(c) $\alpha = 0.5$

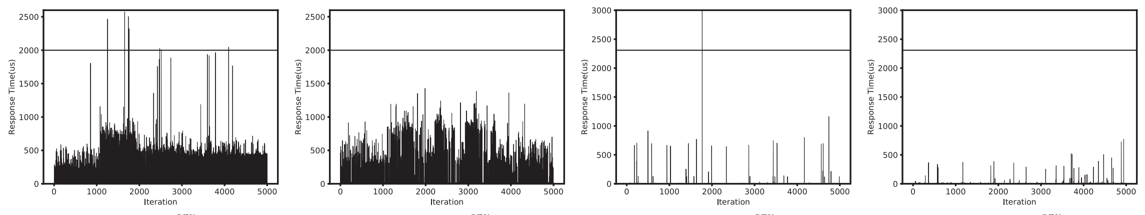
Figure 8: Simulation result

created by the Unifast algorithm[26] leveraging the predefined total utilization of task sets. (3) D_i is calculated by $C_i + (P_i - C_i)\alpha$. The α is the predefined parameter which represents the bias of the D_i between C_i and P_i. It cannot be bigger than 1 or less than 0 because D_i is always bigger than P_i and less than C_i. The slicing overhead O_{i,sc_i} is determined by $0.02 \cdot C_i \cdot sc_i$.

We generate 10^4 task set for each utilization, which has five tasks, and the total utilization U is set in increments of 0.05 within the range of [0.1, 1.0). To show the effectiveness of proposed algorithm, the schedulability of preemptive EDF and non-preemptive EDF without slicing are also measured. All cases use the same input task set. The feasibility test for whole cases uses the exact time analysis presented in [25].

Fig. 8 compares the schedulability of above three approaches for the whole spectrum of the task set utilization of 10^4 task set when α changes. The y-axis of the graph represents the ratio of the number of schedulable task sets out of 10^4 task sets in each case and the x-axis represents total utilization of task set.

In Fig. 8(a), α is 1.0 and it makes all tasks to have implicit deadline. Since preemptive EDF is always schedulable when all tasks have an implicit deadline and total utilization of task set is less than or equal to 1[15], it schedules all task sets in this case. However, the schedulability of task set is decreased as total utilization



(a) Response time of $\tau^{gpu}_{1,5}$ before slicing (b) Response time of $\tau^{gpu}_{1,5}$ after slicing (c) Response time of $\tau^{gpu}_{1,30}$ before slicing (d) Response time of $\tau^{gpu}_{1,30}$ after slicing

Figure 9: Response time of GPU segments before and after slicing

is increased in the non-preemptive EDF case because priority inversion occurs. This tendency is getting worse when relative deadlines are tighter. Relative deadlines are in the 3/4 point and center point between the execution time and the minimum inter release time in Fig. 8(b) and (c), respectively. In these cases, relative deadlines can be less than minimum inter release time, and it makes task set cannot always be scheduled when using preemptive EDF although total utilization is less than or equal to 1. The smaller α sets the tighter deadline so it makes more hard to schedule the task set. The maximum difference of schedulability between the proposed approach and preemptive EDF is 7.1%. Their subtle difference is caused by slicing overhead, and it can be reduced by using a more efficient task slicing method. The maximum improvement on schedulability between sliced and not sliced task set when using non-preemptive EDF is 73.7%. This result proves the fact that the theoretical maximum schedulability of the task set slicing algorithm is the same as preemptive EDF.

6.2 Implementation Results

A real workload experiment also has been conducted to verify practicality. Because any real-time scheduler is not supported on GPU, we implemented a custom software level non-preemptive EDF scheduler for the experiment. Scheduling is performed by requesting the scheduling to the scheduler when the task reaches the start point of a GPU segment and waiting until the task receives the authority. For reducing the scheduling overhead as much as possible, scheduling was conducted based on polling, and shared memory was used to communicate between tasks and the scheduler.

We used five real tasks which are used in autonomous driving: (1) τ_1: LiDAR sensor data based localization module, (2) τ_2: LiDAR sensor data based object detection module, (3) τ_3: darknet[27]-based vision object detection module. Same modules supported in Autoware[28] is used, but code lines for the GPU scheduling and slicing are added. The P_i and D_i of each task were set to the frequency of sensor used in it, and $C_{i,k}^{gpu}$ was determined by profiling the WCET of each task. All $D_{i,k}^{gpu*-}$ was defined by prorating the D_i depending on the WCET of each segment.

The number of GPU segment in τ_1, τ_2, τ_3 is 39, 21, 483, respectively. Total 395,577 GPU segment set candidates are generated, and 35,131 candidates cannot pass the feasibility test before slicing. The maximum slice count for each GPU segment was searched, and slicing for the GPU segment was conducted by splitting the GPU operation into several pieces. After slicing, all segment sets passed the feasibility test.

Fig. 9 shows the measured response time of $\tau_{1,5}^{gpu}$, $\tau_{1,30}^{gpu}$. In each graph, the x-axis is the iteration of each job, and the y-axis is its corresponding response time. The horizontal line in the graph represents the relative deadline of each GPU segment. All tasks are executed at the same time to measure the response time includes priority inversion. As can be seen in Fig. 9(a) and (c), deadline misses have occurred when the slicing algorithm is not conducted. The average response time is lower than the relative deadline, but sometimes it has relatively large peak values. This situation seems to occur when the GPU segment, which has a large WCET, creates priority inversion. Fig. 9 shows the effectiveness of slicing algorithm. No deadline miss occurs after slicing because segments that cause peak value were sliced into several pieces to reduce priority inversion.

7 Conclusion

In this paper, we consider the GPU as a non-preemptive uniprocessor and present a conditionally optimal task set slicing algorithm that enhances the real-time schedulability of GPUs. For this, we derive the essential properties necessary to search the slice count from the feasibility test and define a term called inversion tolerance. Using these properties, we proposed a three-step task set slicing algorithm and proved that it is always optimal under the condition that the slicing overhead is proportional to the slice count. In addition, we prove the effectiveness of the proposed approach through simulation.

To the best of our knowledge, the proposed algorithm is the first way that considers a single GPU as a non-preemptive uniprocessor and transforms the task set to improve schedulability while guaranteeing real-time. One of the significances of our approach is that it can be applied to all types of GPUs.

In the future, we plan to apply our approach to practical application systems such as autonomous driving so that they can be driven on embedded boards with limited resources through optimization. In addition, we plan to expand into a way to utilize GPUs more efficiently for multiple GPUs.

References

[1] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in gpgpus. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296. IEEE, 2012.

[2] Jianlong Zhong and Bingsheng He. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1522–1532, 2013.

[3] Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: A preemptive execution system for gpgpu computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97. IEEE, 2015.

[4] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–16, 2017.

[5] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[6] Sreepathi Pai, Matthew J Thazhuthaveetil, and Ramaswamy Govindarajan. Improving gpgpu concurrency with elastic kernels. *ACM SIGARCH Computer Architecture News*, 41(1):407–418, 2013.

[7] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 119–130, 2015.

[8] Konstantinos Menychtas, Kai Shen, and Michael L Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. *ACM SIGARCH Computer Architecture News*, 42(1):301–316, 2014.

[9] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 109–120, 2014.

[10] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott Brandt. Gdev: First-class {GPU} resource management in the operating system. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 401–412, 2012.

[11] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, pages 485–498, 2013.

[12] Kaibo Wang, Xiaoning Ding, Rubao Lee, Shinpei Kato, and Xiaodong Zhang. Gdm: Device memory management for gpgpu computing. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):533–545, 2014.

[13] Nicola Capodieci, Roberto Cavicchioli, Paolo Valente, and Marko Bertogna. Sigamma: Server based integrated gpu arbitration mechanism for memory accesses. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems*, pages 48–57, 2017.

[14] Shinpei Kato, Karthik Lakshmanan, Aman Kumar, Mihir Kelkar, Yutaka Ishikawa, and Ragunathan Rajkumar. Rgem: A responsive gpgpu execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66. IEEE, 2011.

[15] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[16] Joseph Y.-T. Leung and M.L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[17] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 182–190, 1990.

[18] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science*, 118(1):3–20, 1993.

[19] Ismael Ripoll, Alfons Crespo, and Aloysius K Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1):19–39, 1996.

[20] Laurent George, Paul Muhlethaler, and Nicolas Rivierre. *Optimality and non-preemptive real-time scheduling revisited*. PhD thesis, INRIA, 1995.

[21] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. *ACM SIGARCH Computer Architecture News*, 42(3):193–204, 2014.

[22] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[23] Zhen Lin, Lars Nyland, and Huiyang Zhou. Enabling efficient preemption for simt architectures with lightweight context switching. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 898–908. IEEE, 2016.

[24] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, and Aingara Paramakuru. Deadline-based scheduling for gpu with preemption support. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 119–130. IEEE, 2018.

[25] Laurent George, Nicolas Rivierre, and Marco Spuri. *Preemptive and non-preemptive real-time uniprocessor scheduling*. PhD thesis, Inria, 1996.

[26] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[27] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[28] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles

with embedded systems. In *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pages 287–296. IEEE, 2018.

요약(국문초록)

본 논문은 GPU의 실시간성 보장 및 더 나은 스케줄링 가능성을 위한 조건부 최적 슬라이스 카운트 탐색 알고리즘을 제안한다. 근래 딥러닝의 발전으로 인해 GPU의 중요성이 커지고 있음에도 불구하고, GPU를 실시간으로 활용하기 위한 기술들은 아직 부족한 실정이다. 본 논문은 GPU를 단일 프로세서로 가정하고 비선점형 EDF를 GPU 커널의 스케줄링에 사용한다. 또한 GPU task set을 스케줄링 가능하게 만드는 슬라이스 카운트 탐색 기법을 통해 비선점형 단일 프로세서로의 가정으로 인한 스케줄링 가능성 저하 문제를 해결한다.

주요어 : 실시간 시스템, GPU, 태스크 슬라이싱
학 번 : 2019-28371