TOWARDS SCHEDULING HARD REAL-TIME IMAGE PROCESSING TASKS ON A SINGLE GPU

Vladislav Golyanik$^{1,2}$, Mitra Nasri$^3$ and Didier Stricker$^{1,2}$ *

$^1$University of Kaiserslautern $^2$Department Augmented Vision, DFKI $^3$Max Planck Institute for Software Systems (MPI-SWS)

ABSTRACT

Graphics Processing Units (GPU) are becoming the key hardware accelerators in the emerging image processing applications such as self-driving cars and mobile augmented reality systems. As GPUs execute launched workloads non-preemptively, their usage in safety-critical systems with hard real-time constraints is impeded. The existing solutions for scheduling real-time tasks on a single GPU focus on soft real-time systems. In this paper, we consider real-time systems with a single dedicated GPU handling sporadic tasks with hard deadlines and propose a scheduling approach based on time division multiplexing called the GPU-TDMh — a lightweight middleware framework located between the application and the GPU driver layers. We evaluate the proposed approach on a matrix multiplication benchmark on a heterogeneous platform. The experiments demonstrate the effectiveness of our method as well as superiority over the non-preemptive online scheduling policies.

Index Terms— Graphics Processing Units, Parallel Processing, Real-time, Scheduling, Time Division Multiplexing

1. INTRODUCTION

Nowadays, Graphics Processing Units (GPU) are becoming indispensable hardware accelerators extensively used in image analysis applications with real-time constraints such as tracking and mapping [1], 3D reconstruction [2], angiography treatment [3], gesture recognition [4], advanced driver assistance systems [5] and self-driving technologies [6, 7]. The latter represent complex autonomous systems which track objects around a car and react towards potential dangers. All processing steps must fulfill tight timing constraints since system safety highly depends on the timeliness of the actions.

A GPU is managed by an operating system as an input/output device. If a workload is dispatched to a GPU device, the execution cannot be interrupted until the workload is finished [8] — GPUs of recent generations are mostly non-preemptible devices. For instance, if a high priority task with a short deadline cannot start execution on a GPU due to resource occupation by some low priority task, it may miss the deadline [9]. Accordingly, scheduling of periodic real-time tasks on a GPU is an inherently non-preemptive scheduling problem, known to be strongly NP-hard [10].

Many image processing applications are well parallelizable and support flexible split sizes [11]. The existing widespread scheduling algorithms such as Earliest Deadline First (EDF) or Rate Monotonic (RM) preempt a job (an instance of a task) as soon as a high priority job is released in the system. Thus, if EDF or RM schedule jobs on a GPU, they will cause execution segments of different lengths. Besides, each job will be preempted a different number of times which excludes the option of the compile-time splitting. For the same reason, many other real-time scheduling algorithms such as limited preemption EDF [12] or fixed-priority scheduling with floating preemption model [13] cannot be readily adopted to schedule tasks on a GPU.

In this paper, we propose a scheduling framework based on the notion of Time Division Multiplexing (TDM) — GPU-TDMh (“h” stands for hard real-time constraints). The central component of our approach is a reservation server which activates periodically. During each activation, it schedules one execution segment from each task having a pending job in the system. Thus, it provides timing isolation and does not allow the tasks to affect schedulability of the other tasks. In GPU-TDMh, hard deadlines are guaranteed in the parameter assignment phase.

2. RELATED WORK

TimeGraph developed by Kato et al. [14] is a pioneering driver level framework for scheduling soft real-time tasks on GPUs. It maintains a priority queue and launches the tasks non-preemptively based on priorities. A user-level scheduling solution involving splitting of memory operations was proposed in [15]. In the successor Preemptive Kernel Model [9], it is possible to split memory copies and kernel executions. Steinberger et al. [16] set the split size to the smallest possible launchable segment on a GPU, i.e., a threads block. In [17], a hardware level GPU extension with preemption support is proposed. However, the abovementioned approaches cannot guarantee hard deadlines. In contrast, GPU-TDMh considers the problem of scheduling hard real-time tasks.
Several works consider systems with multiple GPUs [5, 16, 4]. Among them, GPUSync [5] is the most comprehensive framework of this kind. It supports simultaneous data transmission and kernel execution, deterministic task migration among GPUs and implements GPU interrupts through exception handling. GPUSync can be set up to guarantee all deadlines if the workload properties are known in advance. Recently, Otterness et al. [18] indicated possible future extensions to GPUSync for improved schedulability of image processing applications (co-schedulability). As GPUSync is tightly integrated into the LITMUS-RT operating system, it may not be easily portable to other target platforms. In contrast, our lightweight solution is based on time division multiplexing implemented using a reservation server. Since a single server handles all tasks, our approach can also be seen as a Periodic Resource Model (PRM) [19]. The only requirement to the operating system is the support of timer interrupts. As the core contributions of this paper, GPU-TDMH supports hard real-time constraints, is simple to implement and designed with image processing applications in mind.

3. THE PROPOSED FRAMEWORK

In this section, we describe our solution and derive parameter assignment method guaranteeing schedulability of a task set. Although GPU is a non-preemptible device, it is possible to split a kernel into smaller segments and execute them non-preemptively on a GPU.

3.1. Background and System Model

We consider a system with one GPU and a fixed set of associated real-time tasks denoted by \( \tau = \{ \tau_1, \tau_2, \ldots, \tau_n \} \). The tasks are independent and implemented to run on a GPU. Timing parameters of each task \( \tau_i \) are described by the set of values \((C_i, T_i, D_i, \delta_i)\), where \( C_i \in \mathbb{N} \) is the Worst-Case Execution Time (WCET) of the task executing in isolation on a GPU, \( T_i \in \mathbb{N} \) is the minimum inter-arrival time (period), \( D_i \) is the relative deadline and \( \delta_i \in \mathbb{N} \) is the maximum segmenting overhead. We consider implicit deadlines, i.e., \( D_i = T_i \). Besides, we assume that the tasks are indexed according to their periods in ascending order, i.e., \( T_1 \leq T_2 \leq \ldots \leq T_n \).

\( u_i = C_i / T_i \) denotes utilization of \( \tau_i \) and the system utilization is \( U = \sum_{i=1}^{n} C_i / T_i \). A task set is schedulable by a scheduling algorithm if for each instance of a task (a.k.a. job) released in the time interval \( 0 \) to \( \infty \) finishes before its deadline. Liu and Layland [20] showed that \( U \leq 1 \) represents a necessary condition for schedulability of periodic task sets with implicit deadlines. In our system, each task is a CUDA kernel. A kernel associated with a task \( \tau_i \) executes in \( B_i \) blocks with \( R_i \) threads each. The set of all blocks constitutes a grid \( G_i \). \( B_i \) and \( R_i \) define a kernel launch configuration. Using the method introduced in [9], it is possible to partition \( G_i \) into sub-grids and split a kernel into sub-kernels. In this case, the size of a sub-kernel \( S_i \) is determined by the number of blocks in a sub-grid. If the number of sub-kernels is \( k_i \), we obtain \( S_i = B_i / k_i \). In the general case, the number of blocks in sub-kernels may vary. Besides, we make the additional assumption: \( \delta_i \) is a known upper bound of the task splitting overhead — if \( C_i \) is split into \( m_i \) segments, the total execution time of \( \tau_i \) is limited to \( C_i + \delta_i m_i \).

3.2. Parameter Assignment

In this paper, we adopt a time division multiplexing scheme to handle task execution and use a reservation server with the period \( T \) and budget \( C \). Each task receives \( C_i \) units of budget before its deadline. Upon every activation, the server executes a segment of length \( o_i \) from \( \tau_i \) \((1 \leq i \leq n)\). As a result, a part of each released task in the system is executed within every \( T \) units of time. In the rest of this section, we derive \( T \), \( C \) and \( o_i \) parameters so that task set schedulability is guaranteed.

We first derive a lower bound on the number of activations of the reservation server \( m_i(T) \) as a function of \( T \):

\[
m_i(T) = \left\lceil \frac{T_i}{T} \right\rceil - 2,
\]

namely, each job of \( \tau_i \) will receive at least \( m_i(T) \) slots from its release time until its deadline from the reservation server. To guarantee \( C_i \) time units for execution of \( \tau_i \), there must be at least

\[
o_i = \frac{C_i}{m_i(T)} + \delta_i
\]

units of reservation time within each activation of the server. Note that since the execution time of \( \tau_i \) is broken into \( m_i(T) \) slots, at most \( \delta_i \cdot m_i(T) \) time units are wasted in each segment due to the splitting overhead. Taking the sum of \( o_i \) from Eq. (2), we obtain \( C \) as \( \sum_{i=1}^{n} o_i \). In a feasible parameter assignment, the server budget must not exceed its period:

\[
\frac{C}{T} \leq 1 \implies \frac{1}{T} \sum_{i=1}^{n} o_i \leq 1 \implies \frac{1}{T} \sum_{i=1}^{n} \left( \frac{C_i}{m_i(T)} + \delta_i \right) \leq 1.
\]

By replacing \( m_i(T) \) in Eq. (3), we obtain

\[
\frac{1}{T} \sum_{i=1}^{n} \left( \frac{C_i}{T_i/T} - 2 + \delta_i \right) \leq 1.
\]

The only unknown parameter in the Eq. (4) is \( T \), while \( C_i, T_i, \) and \( \delta_i \) are given. Thus, if there exists such \( T \) which satisfies this inequality, schedulability of the task set is guaranteed. However, since \( T \) appears inside the integer ceiling operator in the denominator of the sum in Eq. (4), finding a direct solution for \( T \) is not trivial. Through the following steps, we introduce a way to calculate \( T \). First, the ceiling operator in Eq. (4) is replaced by \( T_i/T \) which makes the denominator smaller and the right-hand side of the inequality larger:

\[
\sum_{i=1}^{n} \frac{C_i}{|T_i/T| - 2} \leq \sum_{i=1}^{n} \frac{C_i}{T_i/T - 2}.
\]
Next, $[T_i/T]$ is replaced by $T_i/T$ in Eq. (4), both denominator and numerator are divided by $T_i$, and, finally, the part containing $\delta_i$ is placed into a separate summation:

$$\sum_{i=1}^{n} \frac{u_i}{1-(2T/T_i^2)} + \sum_{i=1}^{n} \frac{\delta_i}{T} \leq 1.$$  (6)

Task utilization $u_i = C_i/T_i$ is a known value. Although the ceiling function is eliminated, $T$ still appears as a part of the denominator in Eq. (6). The inequality can be solved through approximation by geometric series, as follows:

$$\frac{1}{1-z} = 1 + z + z^2 + z^3 + \ldots,$$  (7)

where $z < 1$. We approximate Eq. (7) by a polynomial of degree 2 given in the general form by $P(z) = a_1 z^2 + a_2 z + a_3$. For $z < 0.7$, the valid upper bound of the series follows

$$P(z) = 4.7z^2 + 1.08$$  (8)

(see our supplementary material for a proof). Next, the substitution $z = 2T/T_i$ in Eq. (8) is performed and applied in $\sum_{i=1}^{n} u_i/(1-(2T/T_i))$ as

$$\sum_{i=1}^{n} u_i \left(4.7 \times \left(\frac{2T}{T_i}\right)^2 + 1.08\right)$$  (9)

which simplifies to

$$18.8 \times T^2 \times \sum_{i=1}^{n} \frac{u_i}{T_i^2} + 1.08 \sum_{i=1}^{n} u_i.$$  (10)

Thus, Eq. (6) can be written as

$$18.8 \times T^2 \times \sum_{i=1}^{n} \frac{u_i}{T_i^2} + \frac{1}{T} \sum_{i=1}^{n} \delta_i + 1.08 \sum_{i=1}^{n} u_i - 1 \leq 0.$$  (11)

By multiplying $T$ on both sides, we obtain

$$18.8 \times T^3 \times \sum_{i=1}^{n} \frac{u_i}{T_i^2} + (1.08 \sum_{i=1}^{n} u_i - 1)T + \sum_{i=1}^{n} \delta_i \leq 0.$$  (12)

Eq. (12) has a form of a depressed cubic equation denoted by

$$T^3 + pT + q = 0, \quad \text{where} \quad p = \frac{1.08 \sum_{i=1}^{n} u_i - 1}{18.8 \sum_{i=1}^{n} (u_i/T_i^2)}, \quad q = \frac{\sum_{i=1}^{n} \delta_i}{18.8 \sum_{i=1}^{n} (u_i/T_i^2)}.$$  (13)

Eq. (13) can be solved using Cardano’s or Vieta’s methods in a closed form. Note that only the real roots which satisfy $T \leq 0.35T_1$ are accepted. Finally, $m_i(T)$ and $o_i$ are derived using Eq. (1) and Eq. (2) given $T$. If none of the resulting $T$ values satisfies the conditions, our approach is unable to find a feasible parameter assignment.

4. EXPERIMENTS

In GPU-TDMh, the server uses a parameter set (see Sec. 3.2) computed at design time and stored in a configuration file, whereby $o_i$ values determine the length of the segments. Our test system consists of the application, middleware, GPU driver and hardware layers (GPU-TDMh occupies the middle-ware layer). Experiments are performed on a system with 32 GB RAM, Intel Xeon processor, the NVIDIA GK110-430-B1 GPU and Ubuntu 16.10 operating system. The performance of our approach is evaluated on a widely used matrix multiplication benchmark for GPU systems [21, 22]. We use a variant of matrix multiplication which involves the device’s shared memory, similar to the one described in [23]. The result is computed in tiles of $32 \times 32$ elements each, whereby each element is handled by a thread; a tile corresponds to a thread block. Thus, our matrix multiplication kernel can be conveniently split into sub-kernels at a granularity level of a single matrix tile. Many widely-used image processing operations (*e.g.*, convolution, median filter, etc. for which it is possible to obtain a safe upper execution bound and the splitting overhead) can be parallelized on a GPU following the same principles. Hereafter, we describe how the task sets with random timing requirements are generated for benchmarking.

In the experiment with the benchmark application, we generate random instances of the matrix multiplication sub-kernel, each with a different WCET and period. Firstly, we create a lookup table containing WCET for the different number of tiles using the WCET function $f_M$ obtained with [24]. To generate a task set, we select periods with uniform distribution from the range $T_i \in [100, 2000]$ milliseconds randomly. Next, we apply the uUniFast algorithm [25] to randomly generate utilization values $U$. Using $T_i$ and $u_i$, we generate the $C_i'$ values for each task (desired WCET values). Finally, we find the closest value to $C_i'$ from the lookup table (more formally, we find the set of parameters $\theta$ of the $f_M$ function so that $f_M(\theta)$ is close to $C_i'$). In this sense, the resulting task set utilization referred to as an expected utilization is close to $U$. For simplicity, we use the $U$ notation for expected utilization. The parameter of our experiment is $U \in [0.35; 0.75]$. Since the hyperperiod can be arbitrarily large in our system, we limit the random task sets to the ones with hyperperiods smaller than $10^9$ milliseconds. For every $U$ value, we perform 100 experiments with different task sets. The whole set of experiments runs roughly 50 hours.

We implemented two widely used real-time scheduling algorithms — EDF and RM — and compared results of our approach with them. However, since GPUs are non-preemptible devices, these algorithms represent non-preemptive EDF and non-preemptive RM. The latter is conceptually close to TimeGraph [14] and RGEM [15] frameworks. We do not consider variants of EDF and RM that schedule single blocks if such variants exist. Though they may resemble preemptive execution on the granularity level of a thread block, they would
utilize only a single streaming multiprocessor at any time.

For every expected GPU utilization value and every scheduling algorithm, the miss ratio, i.e., the ratio of missed jobs to the total number of jobs and actual GPU utilization are measured (see Fig. 1). GPU utilization is defined as a portion of time spent by a GPU for execution of one or several kernels during a unit time interval. Each value on the diagram is obtained by averaging results of 100 runs per each expected utilization value. To measure the actual GPU utilization we employ the \texttt{nvidia-smi} tool launched every 250 milliseconds. The latter value is determined empirically so that the measurements do not harmfully affect task set execution and provide the desired precision. As soon as a task set finishes, the average GPU utilization is computed from the measurements. In our approach, there are no deadline misses for the task sets which are admitted by the parameter assignment step. In average, the measured GPU utilization is 14% lower than the expected one, because the real execution time of the tasks is lower than the estimated WCET. EDF achieves 2.5% miss ratio and 55% schedulability ratio in average. For RM, these values amount to 2.7% and 52% respectively. As expected, EDF performs better than RM.

4.1. Experiments on Synthetic Task Sets

Apart from the experiment with the real task sets, we generate synthetic task sets and further evaluate the performance of our approach. Specifically, we evaluate the schedulability ratio of the proposed parameter assignment method (explained in Sec. 3.2) according to different parameters such as the task set utilization, the number of tasks and the splitting overhead \( \delta \). To generate the task sets, random period values are selected with uniform distribution from the range \([10^2, 10^5]\) milliseconds. We use uUniFast algorithm to generate random utilization values. Finally, \( C_i \) values are computed as \( u_i \times T_i \).

In our experiments, we consider several values of \( \delta \) where \( \forall i : 1 \leq i \leq n, \delta_i \in [0, \delta] \) with uniform chance. We vary \( \delta \) from 0 to 15 with step size 3. Moreover, we consider the effect of the number of tasks in a task set \( n \). To obtain each data point, we generate 5000 random task sets. The resulting diagrams are shown in Fig. 2 and 3 where the horizontal axis is the expected utilization \( U \) and the vertical axis is the schedulability ratio. With the increase in the number of tasks, schedulability ratio decreases because it becomes harder to find a proper parameter set (see Fig. 2). The similar effect occurs when \( \delta \) increases (see Fig. 3). With the increase of the splitting overhead, the reservation server wastes more resources accordingly. Hence, it is not possible to find a feasible parameter assignment for task sets with high utilization.

5. CONCLUSION

We proposed a lightweight scheduling middleware framework for hard real-time sporadic tasks executed on a single GPU — GPU-TDMh. We evaluated our approach on a GPU matrix multiplication benchmark as well as on synthetic task sets. If the parameters were successfully assigned, no deadline misses occurred, while non-preemptive EDF and non-preemptive RM could not schedule about 50% of those task sets. We evaluated the effects of a different number of tasks, splitting overhead and task set utilization. Our method is sustainable towards release jitter and can handle sporadic releases. Future work will address the case with dynamic GPU global memory allocation in augmented reality systems.
6. REFERENCES


