STGM: Spatio-Temporal GPU Management for Real-Time Tasks

Sujan Kumar Saha[1,2], Yecheng Xiang[1], and Hyoseung Kim[1]
[1]University of California, Riverside
[2]University of Florida
sujansaha@ufl.edu, yxian013@ucr.edu, hyoseung@ucr.edu

***Abstract*—Graphics Processing Units (GPUs) have been considered as a promising technology to address the high computational demands of real-time data-intensive applications. Today's embedded processors already offer on-chip GPUs, the use of which can greatly help satisfy the timing requirements of real-time tasks by accelerating their execution. However, existing GPU management schemes either underutilize the GPU due to strictly serialized execution or introduce non-deterministic delay caused by uncontrolled concurrent execution. In this paper, we present a spatial-temporal GPU management framework that controls the allocation and sharing of GPU's internal execution engines, e.g., streaming multiprocessors in Nvidia architectures, with analytical bounds. This approach allows multiple GPU-using tasks to simultaneously execute on the GPU, thereby improving GPU utilization and reducing the worst-case response time. Also, it can improve temporal isolation by allocating a portion of GPU execution engines to tasks for their exclusive use. We have examined the feasibility of our framework on two Nvidia GPUs: GTX970 and AGX Xavier. Experimental results with randomly-generated tasksets indicate that our framework yields a significant benefit in schedulability compared to the existing real-time GPU management approaches.**

I. Introduction

Massive data streams generated by recent embedded and cyber-physical applications pose substantial challenges in satisfying real-time requirements. For example, in self-driving cars, data streams from tens of sensors, such as cameras and LIDARs, should be analyzed in a timely manner. Graphics processing units (GPUs) have been considered as a promising technology to address the high computational demands of real-time data streams. Many of today's embedded processors, such as Nvidia Xavier and NXP i.MX series, already have on-chip GPUs, the use of which can greatly help satisfy the timing challenges of data-intensive tasks by accelerating their execution. The stringent size, weight, power and cost constraints of embedded and cyber-physical systems are also expected to be substantially mitigated by GPUs.

For the safe use of GPUs, much research has been done in the real-time systems community [10, 11, 12, 14, 15, 16, 22, 25]. Many of these schemes limit a GPU to be accessed by *only one* task at a time in order to obtain an analytical bound on the worst case. However, this approach may underutilize the GPU and cause unnecessarily long waiting time when multiple tasks use the GPU. This problem becomes worse in an embedded environment where each machine typically has only a limited number of GPUs, e.g., one on-chip GPU on the latest Nvidia AGX Xavier processors. There are recent studies on exploring the concurrent execution of GPU kernels for real-time systems [2, 20], but it remains unpredictable how much temporal interference may happen among the kernels co-executed on the same GPU execution engine.

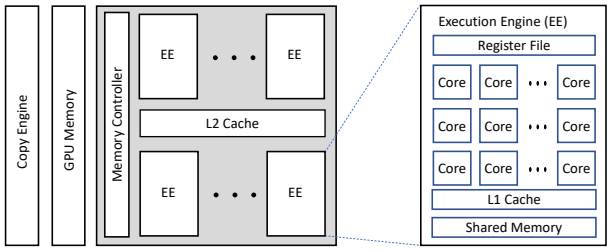


Fig. 1: Overview of GPU structure

In this paper, we present a spatio-temporal GPU management framework to address the aforementioned challenges. The key contribution of this work is in the systematic integration of real-time task scheduling with the partitioning and allocation of GPU's internal execution engines, e.g., streaming multiprocessors on Nvidia GPUs and core groups on ARM Mali GPUs. In our framework, a single GPU is divided into multiple logical units and a fraction of the GPU can be exclusively allocated to each (or a group of) real-time task(s). This approach allows simultaneous execution of multiple kernels on a single GPU, while minimizing timing interference among them. As a proof of concept, we have implemented our framework in a CUDA programming environment for two Nvidia GPUs: GeForce GTX970 and AGX Xavier. Experimental results with randomly-generated tasksets indicate that our framework yields a significant benefit in schedulability compared to the existing approach.

II. Background and Related Work

A. GPU Organization and Kernel Execution

Fig. 1 shows the high-level overview of the internal structure of a GPU. A single GPU consists of multiple Execution Engines (EEs), which are also referred to as Streaming Multiprocessors (SMs) in Nvidia architectures. We will use EEs and SMs interchangeably in the rest of the paper. Each SM has multiple GPU cores. The memory components of an SM, such as register file, L1 cache and shared memory, are shared by all the cores of that SM. Other memory components, such as

*The work was mainly done when the first author was at UC Riverside.

978-1-7281-3197-9/19/$31.00 ©2019 IEEE

L2 cache, GPU main memory and one or more Copy Engines (CEs), are shared among all SMs of the GPU. CEs are used to copy data from CPU memory to GPU memory and vice versa. There also exist several other components, such as instruction buffer, warp scheduler, dispatch units, and texture units, but these are not described in the figure for simplicity.

Nvidia provides CUDA as a GPU programming interface. The general structure of a CUDA program is as follows: (1) memory allocation in GPU memory, (2) data copy from CPU memory to GPU memory, (3) kernel execution on GPU, (4) copy back the results from GPU to CPU memory and (5) free the GPU memory [2, 16]. While launching a kernel, the program provides the thread block and grid dimension information of the kernel to the GPU. The data stream that needs to be processed on a GPU is divided into multiple *logical thread blocks*. The grid consists of all the thread blocks of the kernel, and each thread block consists of multiple threads. In general, each block is processed by a single SM but one SM can process multiple thread blocks if the maximum capacity limit of threads of a SM allows.

The GPU device driver schedules thread blocks on SMs. However, the details of such scheduling for COTS GPUs are not publicly disclosed by manufacturers. Hence, it is hard to predict which thread block will be scheduled on which SM. Despite this difficulty, prior work [2, 9, 21] has experimentally found the following characteristics. First, depending on the number of thread blocks and the size of each thread block, all SMs of a GPU may not be fully utilized. Second, on each SM, more than one thread block can be processed concurrently if the total size of the thread blocks is less than or equal to the capacity of the SM. This has the potential to reduce kernel execution time, but the exact conditions for such concurrent execution within an SM are hard to identify.

B. Related Work

There are several research papers on the use of GPUs in real-time domain. Elliot et al. [10] presented shared resource and container methods for integrating GPU with CPU scheduling in soft real-time systems. GPUSync [11] is a framework based on the k-exclusion locking protocol for real-time multi-GPU systems. The server-based GPU control approach [16] identifies and addresses the limitations of the locking-based approaches, such as busy waiting and long priority inversion. While these approaches focus on predictable GPU control, they do not allow multiple tasks to use the GPU at the same time. As a result, the GPU may be underutilized and there may be a long waiting time for a task to access the GPU.

The work in [3, 14, 25] addresses the non-preemptive behavior of the GPU by splitting the kernel and data copy operations into several sub-parts and by allowing preemption at split points. This approach helps reduce the response time of a high-priority task. Chen et al. [9] proposed a framework, called *Effisha*, to achieve preemptive GPU scheduling without any hardware modification. Even though the above papers contribute to reduce waiting time and improve responsiveness, no one considers simultaneous multi-kernel execution on the GPU to improve utilization while satisfying real-time constraints.

Otterness et al. [20] discussed concurrent multi-kernel execution on Nvidia TX1 and showed that some benchmarks get slowdown in execution compared to when they run independently. Bo et al. [24] proposed a software technique to run a GPU-using task on specific SMs. Janzen et al. [13] presented software-based techniques to partition a GPU among tasks by allocating an exclusive set of SMs to each task. While these techniques are useful to improve GPU utilization, there is no systematic and analytical support to derive the worst-case response time and schedulability of tasks. Hence, it is imperative to investigate the predictable use of multi-kernel execution for real-time systems.

III. System Model

The system we consider is equipped with a multi-core CPU and a GPU. The CPU has N_P cores, where each core is identical to each other and runs at a fixed frequency. The GPU is assumed to follow the architecture described in Sec. II-A. In that GPU, there are N_{SM} SMs. We assume that the GPU has one copy engine (CE), which is typical in many of today's GPUs, and the CE handles copy requests in a *first-come first-serve* basis, following the observations made in [2, 20].

We focus on *partitioned fixed-priority preemptive task scheduling* due to its popularity. For the task model, we consider sporadic tasks with constrained deadlines. Each job of a task consists of *CPU* and *GPU segments*. As their names imply, CPU segments run entirely on the CPU and GPU segments include GPU operations, e.g., data copy from/to the GPU and kernel execution. Once a task launches a GPU kernel, the task may self-suspend to save CPU cycles. The kernel execution time depends on the number of SMs assigned to the task. Specifically, a task τ_i is characterized as follows:

$$\tau_i := (C_i, G_i(k), T_i, D_i, \eta_i, \theta_i)$$

- C_i: The sum of the worst-case execution time (WCET) of CPU segments of each job of τ_i
- $G_i(k)$: The sum of the worst-case duration of GPU segments of each job of τ_i, when k SMs are assigned to τ_i and no other task is using the GPU
- T_i: The minimum inter-arrival time (or period) of τ_i
- D_i: The relative deadline of each job of τ_i
- θ_i: The number of CPU segments of each job of τ_i
- η_i: The number of GPU segments of each job of τ_i

In our system model, $\tau_{i,j}$ and $\tau^*_{i,j}$ are used to denote the j-th CPU and GPU segments of τ_i, respectively. Note that we do not make any assumption about the sequence of CPU and GPU segments. Hence, a task may have two consecutive GPU segments. $G_i(k)$ is assumed to be non-increasing with k, i.e., $G_i(k) \geq G_i(k+1)$. This assumption can be easily met by monotonic over-approximations [1, 17]. The number of SMs assigned to each task is statically determined and does not change at runtime.

We use $G_{i,j}$ to denote the worst-case duration of $\tau^*_{i,j}$ (the j-th GPU segment of a task τ_i). Hence, $G_i(k) = \sum_{j=1}^{\eta_i} G_{i,j}(k)$.

Without loss of generality, each GPU segment is assumed to have *three* sub-segments: (i) data copy to the GPU, (ii) kernel execution, and (iii) data copy back from the GPU. Thus, each GPU segment uses the CE up to *two* times. In this model, more than one consecutive kernels can be represented with multiple GPU segments. $\tau^*_{i,j}$ is characterized as follows:

$$\tau^*_{i,j} := (G^{m_{hd}}_{i,j}, G^e_{i,j}(k), G^{m_{dh}}_{i,j})$$

- $G^{m_{hd}}_{i,j}$: The WCET of miscellaneous operations executed before the GPU kernel in $\tau^*_{i,j}$, e.g., memory copy from the host to the device
- $G^e_{i,j}(k)$: The WCET of the GPU kernel of $\tau^*_{i,j}$ on k SMs
- $G^{m_{dh}}_{i,j}$: The WCET of miscellaneous operations executed after the GPU kernel in $\tau^*_{i,j}$, e.g., memory copy from the device to the host

For the ease of presentation, we may use $G_{i,j}$ to refer to $G_{i,j}(k)$ when k is not needed for explanation. This rule also applies to other GPU-segment parameters, e.g., $G^e_{i,j} \equiv G^e_{i,j}(k)$. We use $G^m_{i,j}$ to represent the sum of $G^{m_{hd}}_{i,j}$ and $G^{m_{dh}}_{i,j}$, i.e., $G^m_{i,j} = G^{m_{hd}}_{i,j} + G^{m_{dh}}_{i,j}$.

The CPU utilization of τ_i is defined as: $U_i = (C_i + G_i)/T_i$, if τ_i busy-waits on the CPU during GPU kernel execution; $U_i = (C_i + G^m_i)/T_i$, where $G^m_i = \sum_{j=1}^{\eta_i} G^m_{i,j}$, if τ_i self-suspends during kernel execution.

IV. Spatio-Temporal GPU Management

A. Framework Design

The goals of the STGM framework are to reduce waiting time for GPU access and to increase GPU utilization, thereby improving taskset schedulability. To achieve these goals, STGM has spatial and temporal management components.

Spatial Management. This component partitions a GPU into SMs and allocates SMs to tasks that require GPU access. The number of SMs assigned to each task is determined by the resource allocation algorithm given in Sec. IV-C. A small code modification is required for each GPU kernel in order to use the spatial management. The code modification of our framework is similar to prior spatial multitasking work [9, 13, 24]. It creates a mapping array to declare the set of SM IDs assigned to the corresponding task and passes that array when launching the kernel. When the thread blocks of the kernel start execution on the GPU, our code modification checks whether the block should run on the current SM or not by looking at the mapping. If the current SM is not valid, i.e., it is not an assigned SM, the block immediately stops execution. Otherwise, the block continues execution. The grid dimension of the kernel is also modified to make sure to run all the blocks on assigned SMs.

Temporal Management. This component controls the temporal behavior of GPU-using tasks. Two cases can occur when multiple GPU-using tasks attempt to access the GPU at the same time with their assigned SMs. First, a task does not have any SM shared with other tasks. In this case, the kernel of that task can start execution on its SMs as soon as data copy is done. Second, a task has at least one SM shared with other

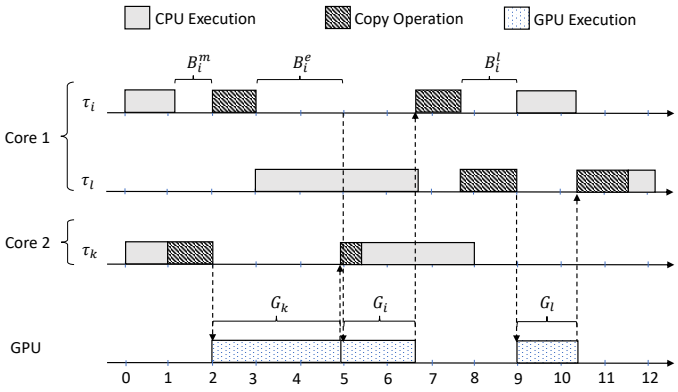


Fig. 2: Example schedule of GPU using tasks experiencing three types of blocking time in self-suspending mode

tasks. In order to provide timing predictability, that task needs to wait until all previously-launched kernels with shared SMs are finished. This is because multiple kernel execution on the same SM may introduce unpredictable delay, as discussed in Sec. II. Here, our framework manages the execution order of such tasks with shared SMs in a FIFO manner, following the default behavior of Nvidia GPUs [20]. Hence, if a task has a shared SM regardless of its priority, it has to wait for the completion of all tasks in the FIFO queue for GPU access, and the waiting time is bounded by our analysis given in Sec. IV-B.

To minimize interference during GPU segment execution, we adopt the *priority-boosting* mechanism, which is widely used for real-time synchronization protocols and predictable shared resource access [7, 16, 18, 23]. Specifically, a task τ_i's priority is increased to the highest-priority level when τ_i begins its GPU segment, and it is reverted back to τ_i's original priority when τ_i finishes that GPU segment. In this way, no CPU segments of other tasks allocated to the same CPU core can preempt τ_i during the interval of τ_i's GPU segment. During kernel execution, GPU-using tasks may either *busy-wait* or *self-suspend*. This is configurable in many GPU programming environments, such as CUDA and OpenCL.

B. Schedulability Analysis

As our framework supports self-suspension and busy-waiting modes, we describe the schedulability analyses for both modes in the following.

1) Self-suspension Mode: If self-suspension is used, the WCRT of τ_i is upper-bounded by the following recurrence:

$$W^{k+1}_i = C_i + G_i + B_i + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \left\lceil \frac{W^k_i + (W_h - (C_h + G^m_h))}{T_h} \right\rceil (C_h + G^m_h) \quad (1)$$

where C_i is the CPU computation time of τ_i, G_i is the total GPU segment time of τ_i, B_i is the total blocking time caused by GPU access, $\mathbb{P}(\tau_i)$ is the set of tasks running on the same CPU core as τ_i, and π_i represents the priority of τ_i. Note that Eq. (1) is an extension of the response time test for general self-suspending tasks proposed by Bletsas et al. [4].

In our framework, the blocking time B_i can be decomposed into three terms: (i) B^m_i, the blocking time from GPU data

copy and miscellaneous operations in GPU segments, (ii) B_i^e, the blocking time from kernel execution, and (iii) B_i^l, the blocking time from priority inversion. Hence, B_i is:

$$B_i = B_i^m + B_i^e + B_i^l \tag{2}$$

Fig. 2 shows an example scenario of the three blocking times that a task can have in its total response time. Here, τ_i and τ_l are running on Core 1 with τ_i having higher priority than τ_l. τ_k is running on Core 2. All tasks are in self-suspension mode. The CPU to GPU copy operation of τ_i is delayed by the copy operation of τ_k as we assume that there is only a single CE in the GPU. This delay is denoted by B_i^m. Then, the kernel execution of τ_i is delayed due the kernel execution of τ_k assuming that τ_i and τ_k have shared SMs. This delay is captured by B_i^e. After the completion of τ_i's GPU segment, τ_i can have blocking time B_i^l during CPU segment (before the start of the last segment in the figure) as τ_l is running its copy operation, which may require CPU intervention, with the highest priority given by the priority boosting of our framework. This delay is denoted by B_i^l.

Lemma 1. *The blocking time from a sub-segment for data copy and miscellaneous operations in the j-th GPU segment of τ_i is upper-bounded by:*

$$B_{i,j}^m = \sum_{\tau_u \neq \tau_i \wedge \eta_u > 0} \max_{1 \leq w \leq \eta_u} G_{u,w}^{m*} \tag{3}$$

where $G_{u,w}^{m*} = \max(G_{u,w}^{m_{hd}}, G_{u,w}^{m_{dh}})$.

Proof. As there is only one CE, $G_{u,w}^{m*}$ is taken by the maximum between host to device copy time and device to host copy time. If τ_u has ω GPU segments, each segment can access the GPU one at a time. Thus, we can take the maximum copy operation time of all the segments of τ_u. Also, the CE is assumed to handle copy requests in a FIFO manner (see Sec. III). Therefore, τ_i has to wait for all the copy operations in the worst case (the summing term). ■

As there are at most two accesses to the CE in one GPU segment, B_i^m is given by:

$$B_i^m = \sum_{1 \leq j \leq \eta_i} 2 \cdot B_{i,j}^m \tag{4}$$

Lemma 2. *The blocking time from any kernel execution in the j-th GPU segment of a task τ_i is upper-bounded by:*

$$B_{i,j}^e = \sum_{\tau_u \neq \tau_i \wedge \mathbb{S}(\tau_u) \cap \mathbb{S}(\tau_i) \neq \emptyset} \max_{1 \leq w \leq \eta_u} G_{u,w}^e \tag{5}$$

where $\mathbb{S}(\tau_i)$ is the set of SM IDs assigned to τ_i. If τ_i does not share its assigned SMs with any other tasks, then $B_{i,j}^e = 0$.

Proof. GPU kernels are executed in an in-order and non-preemptive manner. Hence, in the worst case, each of the tasks using the same SM as τ_i may have requested its longest kernel execution earlier than τ_i. Eq. (5) captures this worst case. ■

The total blocking time from kernel execution, B_i^e, is the summation of $B_{i,j}$ for all segments of τ_i.

$$B_i^e = \sum_{1 \leq j \leq \eta_i} B_{i,j}^e \tag{6}$$

Lemma 3. *The priority inversion blocking imposed on the j-th CPU segment of a task τ_i is bounded by:*

$$B_{i,j}^l = \sum_{\tau_u \in \mathbb{P}(\tau_i) \wedge \pi_u < \pi_i \wedge \eta_u > 0} \max_{1 \leq w \leq \eta_u} G_{u,w}^{m*} \tag{7}$$

Proof. Before the start of τ_i's first CPU segment or whenever τ_i suspends for kernel execution, the GPU segment of each lower-priority task on the same CPU core can have a chance to block τ_i due to the priority boosting of our framework. The amount of blocking from each lower-priority task τ_u is at most $\max G_{u,w}^{m*}$, because after this time, τ_u either suspends for its own kernel execution or recovers its original priority. In the worst case, all lower-priority tasks can cause this blocking to the CPU segment of τ_i, which is captured in the equation. ■

The total priority-inversion blocking B_i^l is given by:

$$B_i^l = \sum_{1 \leq j \leq \theta_i} B_{i,j}^l \tag{8}$$

Note that θ_i (the number of CPU segments of τ_i) is used in the summing term instead of η_i because this blocking can happen for CPU segments, rather than GPU segments that execute with boosted priority [6, 19].

2) Busy-waiting Mode: If tasks are in busy-waiting mode, a simple variant of the traditional response time test is used to upper-bound the WCRT of a task τ_i:

$$W_i^{k+1} = C_i + G_i + B_i + \sum_{\tau_h \in \mathbb{P}(\tau_i) \wedge \pi_h > \pi_i} \lceil \frac{W_i^k}{T_h} \rceil (C_h + G_h + B_i) \tag{9}$$

where B_i is the blocking time caused by GPU access. The blocking time B_i is computed as follows:

$$B_i = B_i^m + B_i^e + B_i^l \tag{10}$$

B_i^m and B_i^e are the same as those in the self-suspension mode. Before the start of τ_i's first CPU segment, the GPU access segments of lower-priority tasks on the same CPU core can block τ_i, which is computed by:

$$B_{i,j}^l = \sum_{\tau_u \in \mathbb{P}(\tau_i) \wedge \pi_u < \tau_i \wedge \eta_u > 0} \max_{1 \leq w \leq \eta_u} G_{u,w} \tag{11}$$

Note that in this equation, $G_{u,w}$ is used instead of $G_{u,w}^{m*}$ because there is no suspension in GPU segments. The total priority inversion blocking B_i^l in busy-waiting mode is:

$$B_i^l = B_{i,j}^l \tag{12}$$

This is because once τ_i starts execution, there is no chance for lower-priority tasks to block τ_i in busy-waiting mode.

C. Resource Allocator

The resource allocation algorithm given in Alg. 1 assigns SMs to each GPU-using task and allocates tasks to CPU cores. The algorithm is based on the worst-fit decreasing (WFD) heuristic for task allocation to balance the load across CPU cores. For SM allocation, the goal of this algorithm is to minimize interference potentially caused by shared SMs among GPU-using tasks, thereby improving taskset schedulability. If there is any task unschedulable due to long GPU execution time with less SMs, more SMs are allocated to that task.

Algorithm 1 SM-aware Task Allocation Algorithm

Require: Γ: a taskset, N_p: Number of CPU cores, N_{SM}: Total number of SM in GPU, P: set of CPU cores (i.e., $|P| = N_p$)
Ensure: N_i: Number of SMs for each task $\tau_i \in \Gamma$, S_i: SM indices for each task $\tau_i \in \Gamma$, Γ_p: a taskset allocated to a CPU core p, U_p: Utilization of tasks in Γ_p if schedulable and ∞ otherwise.

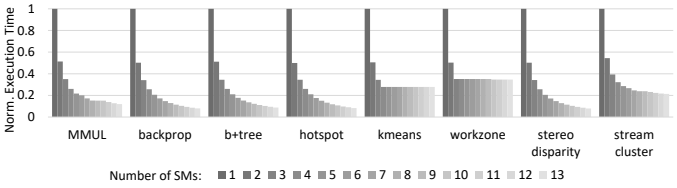
```
 1: for all τ_i ∈ Γ do
 2:     if η_i > 0 then /* GPU-using task */
 3:         N_i ← 1
 4: for p ← 1 to N_p do
 5:     U_p ← 0; Γ_p ← ∅
 6: /* SM Allocation */
 7: sm_idx ← 0
 8: for all τ_i ∈ Γ do
 9:     if η_i > 0 then /* GPU-using task */
10:         S_i ← ∅
11:         for k ← 1 to N_i do
12:             S_i ← S_i ∪ {sm_idx}
13:             sm_idx ← (sm_idx + 1) mod N_SM
14: for all τ_i ∈ Γ in decreasing order of utilization do
15:     for p ∈ P in increasing order of utilization do /* WFD */
16:         if 1 − U_p ≥ C_i/T_i and τ_i satisfies Eq. (1) or (9) then
17:             U_p ← U_p + C_i/T_i
18:             Γ_p ← Γ_p ∪ τ_i
19:             Mark τ_i schedulable
20:             break
21: if all tasks in Γ schedulable then
22:     return {N_i, S_i, Γ_p, U_p}
23: else if ∃N_i ≤ N_max then
24:     i ← argmax_{∀i:τ_i∈Γ∧η_i>0} (G_i(N_i + 1) − G_i(N_i))/T_i
25:     N_i ← N_i + 1
26:     Go to line 7
27: else
28:     return ∞
```

Initially, one SM is given to all GPU-using tasks (lines 1 to 3). In lines 4 and 5, the utilization of each core is set to 0 as no task is assigned to the core yet. Then, the SM IDs are allocated to the GPU-using tasks in lines 8 to 13. Here, if any task has more than one SM, the consecutive SM IDs are allocated to that task. Tasks are assigned to cores according to WFD from lines 14 to 20. If the tasks are schedulable, then the return values are set in line 22. If not all tasks are schedulable, the task that will have the highest benefit in GPU utilization with one extra SM will be assigned one more SM (lines 23 to 24) and the algorithm goes back to the SM allocation phase (line 26). Thus the iteration continues until all tasks are schedulable or the number of SMs for all tasks reaches the maximum number of SMs available in the GPU. As the number of tasks in a taskset is limited, the algorithm will converge after allocating the tasks to cores and if the taskset is not schedulable even after assigning all the available SMs in the GPU, the algorithm will return infinity which indicates that the taskset is not schedulable.

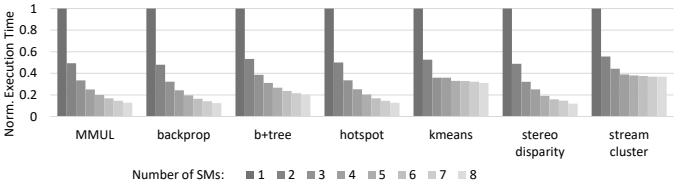
V. Evaluation

A. Implementation

We have implemented a prototype of STGM on two platforms: an x86 machine equipped with a quad-core Intel Core-i7 6700 CPU running at 3.4 GHz and an Nvidia GTX970 GPU and an Nvidia AGX Xavier embedded board. GTX970 has 13 SMs and 2 CEs. AGX Xavier has an integrated Volta GPU with 8 SMs and 16 GB unified memory. We used Ubuntu 16.04 and CUDA 9.0 on the x86, and Ubuntu 18.04 and CUDA 10.0 on the AGX Xavier platform.



(a) GTX970



(b) AGX Xavier

Fig. 3: Kernel execution time w.r.t. the number of SMs

We have selected 8 different GPU benchmarks in the evaluation. `backprop`, `b+tree`, `hotspot`, `kmeans` and `streamcluster` are chosen from the Rodinia GPU benchmark suite [8]. `MMUL` which is a GPU-based computation-intensive matrix multiplication benchmark, `stereodisparity` from Nvidia CUDA sample programs and `workzone` [16] which is a image processing task for self-driving cars to detect the work zones are also chosen for better evaluation. CUDA streams are used to allow asynchronous copy and concurrent kernel execution.

B. Kernel Execution Time

To understand the impact of the number of SMs on kernel execution time, we perform an experiment using eight GPU benchmarks. Fig. 3 shows the normalized GPU execution time of the benchmarks, with the number of assigned SMs varying from 1 to 13 on GTX970 and from 1 to 8 on Xavier. We observe that the execution time does not decrease exponentially with respect to the increasing number of SMs. Also, the execution time of some kernels becomes plateau after a certain number of SMs is assigned. In such cases, there is no benefit to assign more SMs. Specifically, `kmeans` and `workzone` on GTX970 and `kmeans` and `streamcluster` on AGX Xavier show no significant change in execution time when more than 4 SMs are assigned. This is because they have only a small number of thread blocks and a large number of SMs does not help improve the performance. We conclude that, when executing multiple kernels, a proper SM partitioning is the key to improve GPU utilization.

C. Schedulability with Random Tasksets

We have generated 10,000 random tasksets for each schedulability experiment. Each task in a taskset is generated randomly based on the parameters shown in Table I. As GPU kernel execution time varies with different number of SMs, we use Fig. 3a as the execution time profile. We compare the schedulability results of STGM with other approaches. We consider the self-suspending and busy-waiting mode of

TABLE I: Parameters for taskset generation

Parameters	Min	Max
Number of CPU cores (N_c)	2	4
Number of tasks	$2*N_c$	$4*N_c$
Ratio of GPU-using tasks to CPU-only tasks	0.2	2
Utilization of task	0.05	0.3
Number of GPU segments	1	3
Task Period ($T_i = D_i$)	100ms	500ms
Ratio of CPU segment to GPU segment (C_i/G_i)	0.3	0.7
Ratio of kernel length to GPU segment (G_i^e/G_i)	0.7	0.9

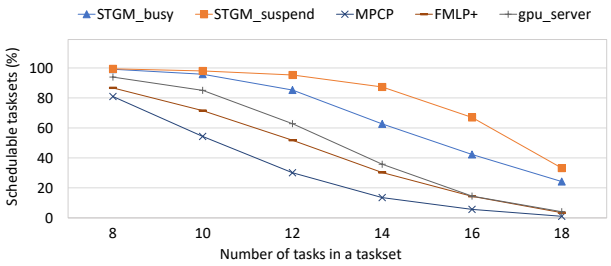


Fig. 4: Schedulability w.r.t the number of tasks in a taskset

our framework (STGM_suspend and STGM_busy). We compare our results with two synchronization-based approaches, Multiprocessor Priority Ceiling Protocol (MPCP) [19, 23] and Flexible Multiprocessor Locking Protocol (FMLP+) [5, 6], and the server-based GPU control approach (gpu_server) [16]. MPCP, FMLP+, and server-based approach allow only a single GPU kernel execution at a time on the GPU.

Fig. 4 depicts the percentage of schedulable tasksets for varying number of tasks. The results show that with more tasks in a taskset, schedulability decreases under all the approaches. STGM_suspend outperforms all other approaches. According to the results, at most 73% more tasksets are schedulable under STGM_suspend compared to MPCP. Compared to busy-waiting mode, self-suspending mode gives higher benefit for both our approach and baseline approach because it yields more CPU time for task execution.

The off-the-shelf GPUs may have different number of SMs. While the number of SMs is another important factor to be considered for schedulability, we evaluate the schedulability with different number of SMs. Fig. 5 presents the percentage of schedulable tasksets with the increasing number of SMs, starting from 1 to 13. When there is only one SM, our approach underperforms the server-based approach. However, as the number of SMs increases, the schedulability of our approach significantly outperforms the others. The schedulability of the synchronization-based and server-based approaches remains almost steady.

VI. Conclusions

The current state of the art GPU management for real-time systems significantly underutilizes GPU resources due to the serialization of GPU kernel execution. The waiting time to access the GPU also is expected to become significant due to the increasing trend of GPU acceleration. In this paper, we presented a spatio-temporal GPU management framework that allows multiple tasks to utilize internal GPU resources simultaneously in a time-analyzable manner. The framework finds

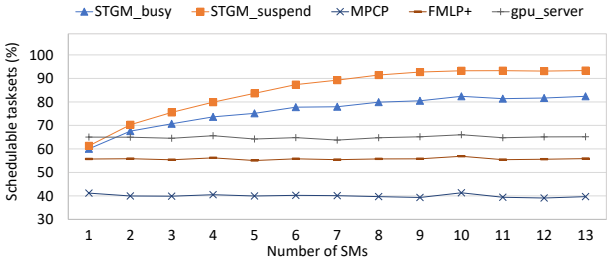


Fig. 5: Schedulability w.r.t the number of SMs

an efficient allocation of GPU resources to tasks and tasks to CPU cores. The schedulability analysis of our framework is presented to bound the maximum blocking time and the worst-case response time of a task. Experimental results indicate that our work improves the schedulability of tasksets significantly compared to the other approaches. As future work, we plan to extend this work to multiple GPUs and further investigate the source of unpredictability in the multi-kernel execution of recent GPUs.

References

[1] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis. Evaluation of cache partitioning for hard real-time systems. In *ECRTS*, 2014.

[2] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *RTSS*, 2017.

[3] C. Basaran and K.-D. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *ECRTS*, 2012.

[4] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical Report CISTER-TR-150713, CISTER, 2015.

[5] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson. A flexible real-time locking protocol for multiprocessors. In *RTCSA*, 2007.

[6] B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems*. PhD thesis, University of North Carolina, Chapel Hill, 2011.

[7] B. Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *ECRTS*, 2014.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[9] G. Chen, Y. Zhao, X. Shen, and H. Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *PPoPP*, 2017.

[10] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48(1):34–74, 2012.

[11] G. Elliott, B. C. Ward, and J. H. Anderson. GPUSync: A framework for real-time GPU management. In *RTSS*, 2013.

[12] S. Hosseinimotlagh and H. Kim. Thermal-aware servers for real-time tasks on multi-core GPU-integrated embedded systems. In *RTAS*, 2019.

[13] J. Janzén, D. Black-Schaffer, and A. Hugo. Partitioning GPUs for improved scalability. In *SBAC-PAD*, 2016.

[14] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *RTSS*, 2011.

[15] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *USENIX ATC*, 2011.

[16] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access control. In *RTCSA*, 2017.

[17] H. Kim and R. Rajkumar. Real-time cache management for multi-core virtualization. In *EMSOFT*, 2016.

[18] H. Kim, S. Wang, and R. Rajkumar. vMPCP: A synchronization framework for multi-core virtual machines. In *RTSS*, 2014.

[19] K. Lakshmanan, D. de Niz, and R. Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.

[20] N. Otterness et al. An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads. In *RTAS*, 2017.

[21] J. J. K. Park, Y. Park, and S. Mahlke. Dynamic resource management for efficient utilization of multitasking GPUs. In *ASPLOS*, 2017.

[22] P. Patel, I. Baek, H. Kim, and R. R. Rajkumar. Analytical enhancements and practical insights for MPCP with self-suspensions. In *RTAS*, 2018.

[23] R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *RTSS*, 1988.

[24] B. Wu et al. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *ICS*, 2015.

[25] H. Zhou, G. Tong, and C. Liu. GPES: a preemptive execution system for GPGPU computing. In *RTAS*, 2015.