# Balancing Energy Efficiency and Real-Time Performance in GPU Scheduling

Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim
University of California, Riverside
ywang665@ucr.edu, mkari007@ucr.edu, yxian013@ucr.edu, hyoseung@ucr.edu

***Abstract*—General-purpose graphics processing units (GPUs) made available on embedded platforms have gained much interest in real-time cyber-physical systems. Despite the fact that GPUs generally outperform CPUs on many compute-intensive tasks in a multitasking environment, high power consumption remains a challenging problem. In this paper, we first analyze the power and energy consumption of GPU kernels scheduled with spatial multitasking, which is found to be advantageous for schedulability in recent studies, and prove that its use, however, degrades energy efficiency even in the latest commercially available embedded GPUs like NVIDIA Jetson Xavier AGX. Then, based on our observations, we propose sBEET, a real-time energy-efficient GPU scheduling framework that makes scheduling decisions at runtime to optimize the energy consumption while utilizing spatial multitasking to improve real-time performance. We evaluate the performance of the proposed sBEET framework using well-known GPU benchmarks and randomly-generated timing parameters on real hardware. The results indicate that sBEET reduces deadline misses up to 13% when the system is overloaded, and also achieves 15% to 21% lower energy consumption when the tasksets are schedulable compared to the existing works.**

## I. Introduction

Nowadays, graphics processing units (GPUs) are already popular due to their outstanding performance. Offloading tasks that require a massive amount of computation and parallelism to the GPUs brings a significant performance improvement to cyber-physical and autonomous applications. Real-time multitasking is an essential prerequisite for developing such GPU-accelerated applications. For example, users can create multiple streams and assign independent kernels to those streams for concurrent kernel execution, in order to achieve speed-up and improve GPU resource efficiency. Power management is one of the major factors for the efficient use of GPUs in an embedded environment. According to [28], GPU power management can bring multiple benefits, such as reducing the energy waste caused by kernel synchronization and resource utilization, improving scalability and reliability through reduced component temperature, and preventing the need for extra cooling.

It is well known that each kernel may not be fully utilizing all the internal computing units of a GPU [7]. In order to better utilize the GPU resources and reduce the waiting time when multiple GPU kernels are sharing a GPU, recent real-time GPU scheduling schemes [19, 21, 31, 32] employ the *spatial multitasking* approach that partitions the GPU into computing units and enables two or more kernels to execute simultaneously on the GPU. While this approach is shown to improve real-time performance and resource efficiency, there has been little consideration of the resulting energy consumption. In fact, as we will discuss in the paper, the energy consumption of GPU kernels scheduled by the previous schemes using spatial multitasking can be much worse compared to the naive approach that executes one kernel at a time. This is due to the lack of power gating at a granularity of computing units in most commercially available GPUs.[1] Hence, when spatial multitasking is used, idle computing units can continue to consume dynamic power as long as at least one computing unit remains actively used by kernels. Meanwhile, kernels with fewer computing units have longer execution time, thereby further increasing energy consumption.

This paper presents sBEET, a scheduling framework that Balances Energy Efficiency and Timeliness of GPU kernels to address the aforementioned challenges on embedded GPUs. This work presents a generic power model to capture the characteristics of dynamic and idle power consumption of GPU kernels, and provides an analysis of GPU energy consumption with spatial multitasking. At runtime, sBEET makes scheduling decisions about the partitioning of computing resources, e.g., *streaming multiprocessors* in NVIDIA GPUs, based on the prediction of energy consumption calculated by the power model. This approach allows simultaneous kernel execution to improve real-time performance while reducing the energy consumption of the GPU.

To evaluate the performance of sBEET, we implemented the framework on an NVIDIA Jetson Xavier AGX platform. Experiments are conducted using randomly-generated tasksets of well-known benchmarks to compare the schedulability and energy consumption of our framework against several representative existing approaches: the default First-Come-First-Serve (FCFS) scheduling of NVIDIA GPUs [9], the fixed-priority Rate Monotonic (RM) scheduling without spatial multitasking, and the scheduling algorithm proposed in [31] that uses spatial multitasking. Experimental results show that sBEET addresses the problem of spatial multitasking by maintaining a similar energy consumption as the non-spatial multitasking approaches while reducing the occurrence of deadline misses significantly. The results also demonstrate that sBEET brings substantial benefits in both real-time performance and energy saving when compared to the spatial multitasking

[1] Power-gating overhead is one of the major obstacles since each execution unit of a GPU tends to idle for shorter periods than break-even time [6].

approach.

In summary, this paper makes the following contributions:

- We derive a power and energy consumption analysis for GPU kernels scheduled with and without spatial multitasking on the GPU, and find that the use of spatial multitasking could result in higher energy consumption.
- We develop a runtime scheduling algorithm that reduces deadline misses of non-preemptive GPU kernels by dynamically adjusting the degree of resource partitioning and improves energy efficiency over the existing spatial multitasking approach.
- Finally, we demonstrate the practical effectiveness of sBEET in real-time performance and energy consumption through a diverse set of experimental scenarios on the latest commercially available embedded GPU platform.

## II. Background and System Model

### A. Background

**NVIDIA Jetson AGX Xavier.** This paper considers the latest embedded GPU platform, NVIDIA Jetson AGX Xavier. The architecture of the System-on-Chip (SoC) used in this platform is illustrated in Fig. 1a. The Xavier SoC features eight 64-bit ARMv8 Carmel CPU cores running at 2265MHz with 128KB instruction and 64KB data L1 caches, 2MB of L2 cache per cluster of two cores, and a 4MB of L3 cache shared by all CPU clusters. The SoC has an integrated 512-core Volta GPU sharing 16GB of 2133MHz memory with the CPU while consuming less than 30 Watts. The GPU includes eight execution engines, also known as *streaming multiprocessors* (SMs), each containing 64 CUDA cores and 8 Tensor cores. Each SM includes a 128KB L1 cache, and all the SMs share a 512KB L2 cache. It also comprises several other computing elements and accelerators such as DLAs, vision accelerator, and video encoder/decoder [1], but we primarily focus on the energy consumption of the GPU component.

**SM Organization and Kernel Execution.** The CUDA programming model provides an abstraction of the GPU architecture that the user can directly interact with. The general steps to execute any CUDA program includes: (i) memory allocation in both CPU and GPU side, (ii) copy the input data from CPU memory to GPU memory, (iii) execute the GPU program, (iv) copy the results from GPU memory to CPU memory, and (v) free the GPU memory [9, 17, 24]. The parts that run on the GPU are known as CUDA kernels, and each kernel is executed by different CUDA threads in parallel. A group of threads is called a thread block, and thread blocks are grouped into a grid. The number of blocks and grids is defined by the user before launching the kernel. The user can also use CUDA streams to achieve parallel execution of multiple CUDA kernels, and each stream manages a FIFO queue for kernel execution. Once a kernel starts on the GPU, its execution cannot be preempted (except by another kernel from a stream with a higher priority). The blocks are distributed onto SMs in a nearly round-robin manner [9, 29] and cannot be migrated between SMs. One SM can run multiple blocks concurrently depending on their resource demands, such as shared memory, the number of threads, and the number of register files, etc. By default, kernels are executed using all the SMs of the GPU on a First-Come-First-Serve (FCFS) basis.

**Power Management in Xavier AGX.** Jetson AGX Xavier has two input voltage rails: SYS_VIN_HV and SYS_VIN_MV, each having a different range of input voltages, i.e., 9V to 19V for SYS_VIN_HV and 5V for SYS_VIN_MV, which the device can switch between for power efficiency. The power consumption of the rails in Fig. 1b can be measured by a built-in power monitor which has a range up to 26V [2]. There are some power management mechanisms related to the circuit design that are used to optimize power efficiency when (part of) the device is detected to be idling:

- Clock-gating: remove the clock signal.
- Power-gating: shut off the power supply to the circuits within the rails.
- Rail-gating: shut off the power supply to the entire rail.

The GPU is both rail-gated and clock-gated, but the details on how the circuits work are not publicly available. Nonetheless, as experimentally confirmed in Section V-B, SM-level power/clock gating does not appear to exist even on the latest Xavier AGX GPU.

### B. System Model and Assumptions

We consider a taskset $\Gamma$ consisting of $n$ real-time non-preemptive periodic tasks with constrained deadlines. We focus on the kernel execution and memory copy operations, and a task $\tau_i$ is characterized as follows:

$$\tau_i := (G_i, T_i, D_i)$$

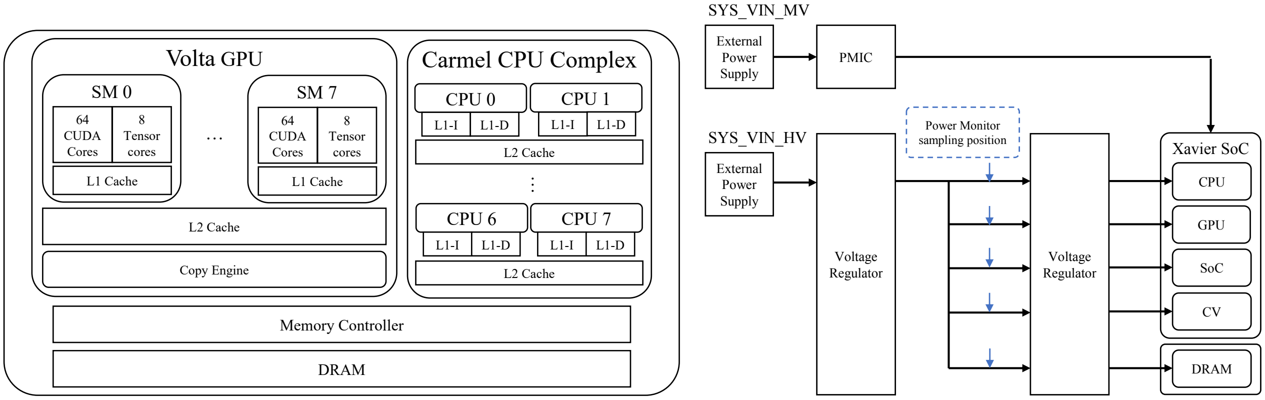
- $G_i$: The cumulative worst-case execution time (WCET) of GPU segments (including memory copies and kernels) of a single job of $\tau_i$. The duration depends on how many SMs are assigned to a particular job.
- $T_i$: the period or the minimum inter-arrival time.
- $D_i$: the relative deadline of each job of $\tau_i$, and is smaller than or equal to the period, i.e., $D_i \leq T_i$.

A task $\tau_i$ consists of a sequence of jobs $J_{i,j}$, where $J_{i,j}$ indicates the $j$-th job of task $\tau_i$. Following the idea of spatial GPU multitasking [19, 21, 31, 32], each job $J_{i,j}$ of the task $\tau_i$ can execute with a different number of SMs exclusively assigned to it. Hence, we use $G_{i,j}(m)$ to represent the WCET of $J_{i,j}$, where $m$ denotes the number of SMs used by $J_{i,j}$. $G_{i,j}(m)$ is given by the sum of the following three parameters:

$$G_{i,j}(m) = G_i^{hd} + G_{i,j}^{e}(m) + G_i^{dh}$$

- $G_i^{hd}$: the worse-case data copy time from the host to the device memory
- $G_{i,j}^{e}(m)$: the worst-case kernel execution time of $J_{i,j}$ when $m$ SMs are assigned to it
- $G_i^{dh}$: the worse-case data copy time from the device to the host memory

The *utilization* of a task $\tau_i$ is defined as the average utilization when different number of SMs are assigned, and it is computed as $U_i = \frac{\sum_{m=1}^{M} U_i(m)}{M}$, where $M$ is the total



(a) Xavier SoC Architecture

(b) Block diagram of Xavier power rails

Figure 1: Architecture and module power rails of NVIDIA Jetson AGX Xavier

number of SMs on the device and $U_i(m) = \frac{G_i(m)}{T_i}$. It is worth noting that $U_i(M) \leq 1$; otherwise, $\tau_i$ is never schedulable regardless of how many SMs are given. The *total utilization* of a taskset is denoted as $U = \sum_{\tau_i \in \Gamma} U_i$. Each job is characterized by an arrival time $r_{i,j}$ and an absolute deadline $d_{i,j} = r_{i,j} + D_i$. Without loss of generality, we assume a discrete-time system where timing parameters can be represented in positive integers.

Based on the kernel execution time $G^e_{i,j}(m)$, each job $J_{i,j}$ can be categorized into either *linear-speedup* or *nonlinear-speedup* job [32]:

- $J_{i,j}$ is a linear-speedup job if $G^e_{i,j}(m)$ is inversely proportional to $m$, i.e., $\forall m(m \leq M), G^e_{i,j}(m) = G^e_{i,j}(1)/m$. This applies to most kernels that typically have many thread blocks ($\gg M$) with reasonable memory demands because when more SMs are assigned, the thread blocks of such kernels can be evenly distributed across the SMs and be executed in parallel.
- $J_{i,j}$ is a nonlinear-speedup job if there exists a case where the speedup is nonlinear to the number of SMs assigned, i.e., $\exists m(m \leq M)$, $G^e_{i,j}(m) > G^e_{i,j}(1)/m$. This happens to kernels that have only a small number of thread blocks or saturate the memory resources of the GPU (e.g., bandwidth, shared memory, registers, etc.) [7].

This categorization will be used for proofs of energy consumption properties in spatial multitasking (Sec. IV-A), but our proposed runtime scheduling algorithm (Sec. IV-B) works regardless of the kernel type.

## III. Related Work

**Temporal Multitasking on GPU.** Some prior works have been done to improve GPU utilization in a time domain. TimeGraph [23] is a real-time GPU task scheduler that assigns temporal budgets to tasks with different priorities, and replenishes the budgets periodically. Elliott et al. [15] modeled the GPU as a mutually-exclusive shared resource and used real-time synchronization protocols to integrate the GPU into real-time multiprocessor systems. Kim et al. [24] proposed a server-based approach to address the busy waiting and long priority inversion problems of the synchronization-based approach.

Temporal multitasking can be divided into two types: preemptive scheduling and non-preemptive scheduling. The aforementioned methods [15, 23, 24] treat GPU tasks as non-preemptive tasks, which allows a task to exclusively use the computing units of the GPU only after the currently-running tasks release the GPU. On the other hand, some other works [10, 22, 42] introduce software-based mechanisms to enable preemptive scheduling of real-time GPU tasks. The key idea is to decompose a long-running kernel into smaller segments so that preemption can happen at the boundaries of these segments. However, regardless of the preemptiveness of GPU tasks, temporal multitasking can suffer from the resource underutilization problem given that each GPU task may not fully utilize all the computing units of the GPU [7]. In addition, GPU tasks can experience a long waiting time if they are scheduled non-preemptively. The use of preemptive scheduling can reduce this waiting time, but the implementation of the software-based mechanisms is not trivial since they require modifications to device drivers and the hardware-level preemptions provided in recent GPUs have only a limited number of priority levels (e.g., only two in the NVIDIA Pascal and Volta architectures [5, 40]).

**Spatial Multitasking on GPU.** Spatial multitasking, also called spatial resource sharing or GPU partitioning, focuses on the fact that multiple tasks can execute simultaneously on different subsets of computing units of the GPU. There are some works done on spatial multitasking [7, 8, 19, 27, 34, 37]. Specifically, Jain et al. [19] proposed to spatially partition computing units as well as on-board DRAM to enable parallel kernel execution, better resource utilization, and performance isolation. However, these works did not pay much attention to the real-time constraints of individual GPU tasks.

Sun et al. [32] proposed algorithms to minimize the makespan of a static schedule of GPU kernels by taking into account the long data transfer duration prior to GPU kernel execution and modeling kernels as moldable parallel jobs. However, the static schedule generated by their algorithms assumes all jobs arrive at the same time with the same period, thereby unsuitable to periodic or sporadic real-time tasks with arbitrary release offsets which are prevalent in cyber-physical systems. Kang et al. [21] focused on mobile latency-sensitive workloads and proposed the spatial resource reservation technique that reserves computing units for high-priority tasks to help reduce the blocking time of foreground applications from background ones. Wang et al. [39] developed a QoS mechanism that allocates resources dynamically to meet the QoS goals of individual GPU kernels. For periodic real-time tasks, Saha et al. [31] proposed spatial-temporal GPU management (STGM) that combines temporal and spatial multitasking to improve taskset schedulability under the Rate Monotonic (RM) policy. The resource allocation algorithm of STGM first assigns the minimum number of SMs to each task, and if any task is unschedulable due to the long execution time caused by a small number of SMs assigned to it, the algorithm gives more SMs to that task. In this way, STGM can reduce potential interference caused by SMs shared with other tasks. However, in Section IV-A, we will analyze the energy consumption of GPU tasks in the presence of spatial multitasking, and such techniques cannot lead to the most energy-efficient schedule.

**Resource Allocation for GPU Energy Saving.** Wang and Ranganathan [38] developed an instruction-level prediction mechanism to save energy by estimating the number of SMs for a given application. Wang et al. [36] proposed power gating strategies to turn off extra resources that are not being used. Since the switching overhead often yields negative energy saving, they ensure that the unused circuits remain off long enough to compensate for the overhead. Hong and Kim [16] put forward an integrated power and performance prediction to improve the GPU energy efficiency by building a resource-based power model and finding the optimal number of SMs for each workload that leads to the highest performance-per-Watt. They also proposed a theoretical method that the unused SMs can be shut off by a power-gating mechanism if it is supported at the circuit level. Aguilera et al. [8] presented QoS-aware dynamic resource allocation and experimentally demonstrated the effectiveness of this method in GPGPU-Sim. Sun et al. [33] proposed a runtime QoS management mechanism that dynamically adjusts SM allocation so that the idle SM can be power gated to reduce energy consumption. Zahaf et al. [41] presented a general model of energy consumption and performance on heterogeneous multi-core processors such as ARM big.LITTLE, and proposed a heuristic approach to reduce energy consumption for soft real-time moldable parallel tasks. Tasoulas et al. [35] categorized GPU workloads according to their resource demands and achieved energy savings in GPGPU-Sim by pairing the workloads and power-gating unused SMs. However, the results cannot be directly applied to real hardware platforms since the implementation of per-SM power-gating is not yet available in today's GPUs.

## IV. sBEET Framework

This section presents the sBEET framework and analysis. We first introduce power and energy analysis, and then propose a scheduling algorithm that makes runtime scheduling decisions and SM allocation.

### A. Power and Energy Analysis

**Power model.** Isci and Martonosi [18] introduced a general framework originally designed for CPU power modeling, which is also widely used as a basis for many GPU simulation works, such as Hong and Kim's model [16] and *GPUWattch* by Leng et al. [26]. It defines the total power consumption $P$ as the sum of idle power $P^{idle}$ from idling cores (SMs in our case), leakage power (static power) $P^s$, and dynamic power $P^d$ from active SMs. Following this approach, the instant power consumption of the GPU at time $t$ can be written as:

$$P = P^s + P^d + P^{idle} \tag{1}$$

$P^d$ is the power consumption required to execute kernels on SMs and depends on the kernel characteristics including memory access and the number of SMs used [16].[2] Hence, $P^d$ can be decomposed into a linear sum of per-SM power consumed by each job. For a set of jobs $\mathrm{J} = \{J_1, J_2, ..., J_n\}$[3] executing simultaneously on the GPU at time $t$, Eq. (1) can be rewritten as:

$$P = P^s + \sum_{i=1}^{n} P_i^d(m_i) + P^{idle}(M - \sum_{i=1}^{n} m_i) \tag{2}$$

where $m_1, ..., m_n$ are the number of SMs being exclusively used by $J_1, ..., J_n$, respectively.[4] $P_i^d(m)$ is the dynamic power consumption of $J_i$ on $m$ active SMs, $P^{idle}(m)$ is the idle power of $m$ inactive SMs, $M$ is the total number of SMs on the GPU. Note that when all the SMs of the GPU are idling ($\sum_{i=1}^{n} m_i = 0$), the GPU is power-gated and there is no power consumption from $P^d$ and $P^{idle}$, i.e., $\sum P_i^d(0) = 0$ and $P^{idle}(M) = 0$. In addition, since the dynamic (and idle) power consumption is linear to the number of active (and inactive) SMs [16], the following conditions hold:

$$P_i^d(m) \propto m, \quad \text{and} \quad P^{idle}(m) \propto m \tag{3}$$

Using $P$, the energy consumption of the GPU for the time period $[t_1, t_2]$ can be computed as follows:

$$E = \int_{t_1}^{t_2} P dt \tag{4}$$

Now let us consider a set of jobs $\mathrm{J} = \{J_1, J_2, ..., J_n\}$ that are *scheduled* on the GPU during $[t_1, t_2]$. Depending on scheduling decisions, some jobs of J may be active at $t \in [t_1, t_2]$ while the others may be inactive. We define a binary indicator $x_i^k(t)$ that returns 1 if the $k$-th SM is actively used by a job $J_i$ at time $t$, and 0 otherwise. Using this, Eq. (4) can be re-written as follows:

$$\begin{aligned} E(t_1, t_2) = \int_{t_1}^{t_2} \Bigg( P^s + \sum_{i=1}^{n} \Big( P_i^d(\sum_{k=1}^{M} x_i^k(t)) \Big) \\ + P^{idle}\Big( M - \sum_{i=1}^{n} \sum_{k=1}^{M} x_i^k(t) \Big) \Bigg) \end{aligned} \tag{5}$$

[2] The dynamic power characteristics of each kernel can be estimated by either measurement-based profiling or analytical methods [16]. We use the profiling approach in our evaluation.

[3] For simplicity, we omit the index $j$ of $J_{i,j}$ since we do not need to refer to individual jobs of the same task.

[4] This means no shared SM between jobs, i.e., at any time instant $t$, $\sum_{i=1}^{n} m_i \leq M$, which is required for simultaneous execution on the GPU with spatial multitasking.

The detailed methods to obtain the above power parameters and to realize SM allocation on commodity GPU hardware will be explained in Section V-A. Based on the above power and energy model, we analyze the energy consumption of a schedule with spatial multitasking in the following theorem.

**Theorem 1.** *The schedule of a job set* J *with spatial multitasking cannot be more energy-efficient than the schedule without spatial multitasking if the jobs in* J *are linear-speedup jobs.*

*Proof.* Consider a job set with two jobs, $\mathrm{J} = \{J_1, J_2\}$, which arrive together at time $t_1$. For this job set, there are two possible schedules that can be obtained with and without spatial multitasking: (a) sequentially executing the first job $J_1$ on $M$ SMs and then the second job $J_2$ on $M$ SMs (w/o spatial multitasking), and (b) simultaneously executing $J_1$ on $m_1$ SMs and $J_2$ on $m_2$ SMs, where $m_1 + m_2 = M$ (w/ spatial multitasking). For convenience, we assume $G_1^e(m_1) \leq G_2^e(m_2)$, i.e., $J_2$ finishes later than $J_1$. To assess the energy efficiency of these two schedules, we consider a time interval $\delta = [t_1, t_2]$ that is long enough to complete job-set execution under both schedules. The energy consumption of the two schedules, $E_a$ and $E_b$, during $\delta$ can be respectively written as below:

$$E_a = P^s \cdot \delta + P_1^d(M) \cdot G_1^e(M) + P_2^d(M) \cdot G_2^e(M) \tag{6}$$

$$\begin{aligned} E_b &= P^s \cdot \delta \\ &+ (P_1^d(m_1) + P_2^d(m_2)) \cdot G_1^e(m_1) \\ &+ (P_2^d(m_2) + P^{idle}(M - m_2)) \cdot (G_2^e(m_2) - G_1^e(m_1)) \end{aligned} \tag{7}$$

Recall that when the execution completes and all SMs are idling, $\sum P_i^d(0) = 0$ and $P^{idle}(M) = 0$ due to power gating.

Using $E_a$ and $E_b$, we now prove the theorem by contradiction. Assume that there exists a case where the schedule with spatial multitasking is more energy-efficient than that without spatial multitasking, i.e., $\exists m_1 \exists m_2, E_a > E_b$. Since we consider *linear-speedup* jobs here, $G_i^e(m_i) = G_i^e(1)/m_i$ where $G_i^e(1)$ is assumed to be a known constant. Then,

$$\begin{aligned} & E_a - E_b > 0 \\ \Leftrightarrow \quad & G_1^e(1) \cdot (\frac{P_1^d(M)}{M} - \frac{P_1^d(m_1)}{m_1}) \\ & + G_2^e(1) \cdot (\frac{P_2^d(M)}{M} - \frac{P_2^d(m_2)}{m_2}) \\ & + P^{idle}(M - m_2) \cdot (G_1^e(m_1) - G_2^e(m_2)) \\ & > 0 \end{aligned} \tag{8}$$

From Eq. 3, for any $m$, $\frac{P_i^d(M)}{M} = \frac{P_i^d(m)}{m}$, so the first two terms are 0 and we can rewrite the Eq. 8 to:

$$\begin{aligned} & E_a - E_b > 0 \\ \Leftrightarrow \quad & P^{idle}(M - m_2) \cdot (G_1^e(m_1) - G_2^e(m_2)) > 0 \end{aligned} \tag{9}$$

That leads to $G_1^e(m_1) > G_2^e(m_2)$. It contradicts to our assumption of $G_1^e(m_1) \leq G_2^e(m_2)$. Thus, the assumption that $E_a > E_b$ is false and the lemma is proved. The same approach can used to prove the case where there are more than two linear-speedup jobs. □

**Lemma 2.** *Theorem 1 does not necessarily hold for nonlinear-speedup jobs.*

*Proof.* By the definition of nonlinear-speedup jobs, $\exists m (m \leq M)$, $G_{i,j}^e(m) > G_{i,j}^e(1)/m$. Hence, $\frac{P_i^d(M)}{M} \neq \frac{P_i^d(m)}{m}$, and we cannot deterministically compare $E_a$ and $E_b$ in Eq. 8. □

Theorem 1 gives an insight that for *linear-speedup* jobs, the spatial multitasking strategy unavoidably causes some SMs to be idling while the GPU is active since GPU power is not SM-gated; therefore, the power consumption of idle SMs affects the overall energy consumption of the schedule which is less energy-efficient than the schedule without spatial multitasking. However, Lemma 2 opens a possibility that for each *nonlinear-speedup* job, there may exist an optimal number of SMs that leads to the most energy-efficient schedule.

**Definition 1.** The energy-optimal number of SMs $m_i^{opt}$ for a task $\tau_i$ is defined as the number of SMs that leads to the lowest energy consumption when it executes in isolation on the GPU during an arbitrary time interval $\delta \geq \max_{m \leq M} G_{i,j}^e(m)$.

The time interval $\delta$ considered for $m_i^{opt}$ is to take into account the impact of idling SM time when $\tau_i$ does not use all SMs. The length of the time interval does not affect the value of $m_i^{opt}$ as long as the interval is greater than or equal to the maximum GPU execution time of any job in $\tau_i$, because once $\tau_i$ completes execution, the GPU is power-gated and only the static power $P^s$ contributes to the total energy consumption.

**Example 1.** Consider a taskset $\Gamma$ with the following two *linear-speedup* tasks on a GPU with $M$ identical computing units. The memory copy operation and GPU execution time of these tasks are listed in Table I. The tasks are running with different CUDA streams, so synchronized memory copy and concurrent kernel execution are possible. An interval of interest [0, 12) is considered for the following two cases: a schedule of the two tasks with and without spatial multitasking.

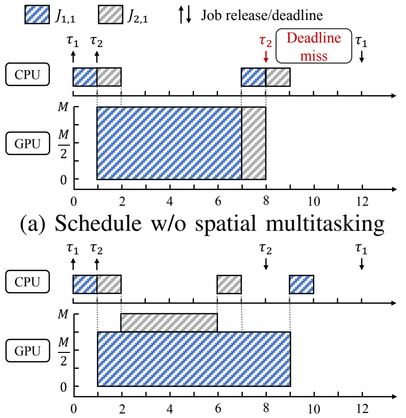
Table I: Taskset in Example 1

| Task | $D_i$ | $G_i^e(M)$ | $G_i^{hd}$ | $G_i^{dh}$ | Offset |
|---|---|---|---|---|---|
| $\tau_1$ | 12 | 6 | 1 | 1 | 0 |
| $\tau_2$ | 7 | 1 | 1 | 1 | 1 |

Fig. 2a shows the schedule without spatial multitasking. Since there are only two tasks and $\tau_1$ arrives earlier than $\tau_2$, any work-conserving scheduling policies would yield the same schedule. At $t = 0$, $J_{1,1}$ is scheduled on the GPU since no other job is ready. After memory copy, it occupies all the GPU cores in [1, 7). The following job $J_{2,1}$ arrives at $t = 1$, but because of the blocking by $J_{1,1}$, $J_{2,1}$'s GPU execution cannot start until $t = 7$ and finally it misses the deadline.

Fig. 2b shows the schedule with spatial multitasking. By running $J_{1,1}$ with $\frac{3 \cdot M}{4}$ SMs, both jobs $J_{1,1}$ and $J_{2,1}$ can be scheduled without any deadline miss. However, the energy consumption of the schedule in Fig. 2b is greater than that in Fig. 2a, and this can be easily derived from Theorem 1.

In summary, the above example suggests that scheduling of GPU jobs with no spatial multitasking can cause a deadline



(a) Schedule w/o spatial multitasking

(b) Schedule w/ spatial multitasking

Figure 2: Scheduling results in Example 1

miss due to the blocking time from an earlier job. While the use of spatial multitasking addresses this problem, it could increase energy consumption since the time for all GPU units being idle is reduced, as given by the above analysis. Motivated by this example, our goals are: (i) to minimize deadline misses by exploiting the spatial multitasking technique, and (ii) to maximize the opportunity to reduce energy consumption by running each job with the optimal number of SMs ($m_i^{opt}$) whenever possible.

### B. Scheduling Framework

sBEET consists of one server and multiple worker threads. Similar to MPS [4], the server receives the jobs of GPU tasks so that they share a single CUDA context, and dispatches the jobs to the worker threads for execution with separate CUDA streams on the GPU. In this way, sBEET enables spatial multitasking for parallel kernel execution when it is needed, and the decision on when to use spatial multitasking is made by our scheduling algorithm presented later. One of the important design issues is the number of worker threads that determines how many kernels can run concurrently on the GPU. In this work, we limit the number of workers to *two* due to the following reasons: (i) the use of more workers can lead to more SM going idle at different times, which increases energy consumption as discussed in Section IV-A (also see results in Fig. 10); (ii) more workers mean more combinations of SM allocation available for each kernel launched by each worker, and the overhead from increased computational complexity may become unacceptable for the runtime framework running on embedded platforms; (iii) more workers may increase contention on shared resources as reported in [7]; and (iv) based on our observation, creating more workers does not necessarily contribute to reducing deadline misses on embedded GPUs.

During the initialization phase, the server creates two worker threads as well as two CUDA streams, and each worker is bonded to one of the streams. When a job is offloaded to the worker, it runs on the corresponding CUDA stream. The workers communicate with each other via a global shared data structure which is also created during the initialization phase.

**Algorithm 1** Runtime Scheduler

```
Input: $J_{i,j}$: the first job in the ready queue
Input: $S_{avail}$: the set of currently available SMs
procedure SCHEDULER($J_{i,j}$, $S_{avail}$)
    if $|S_{avail}| = M$ then                                   ▷ GPU is idling
        $S_{i,j}^{cfg} \leftarrow$ ALLOCATION($J_{i,j}$, $nullptr$)     ▷ Alg. 2
        if $S_{i,j}^{cfg} \neq \emptyset$ then
            Execute $J_{i,j}$ with $S_{i,j}^{cfg}$; remove $J_{i,j}$ from ready queue
        end if
    else if $0 < |S_{avail}| < M$ then
        $J_{q,r} \leftarrow$ currently running job
        $S_{i,j}^{cfg} \leftarrow$ ALLOCATION($J_{i,j}$, $J_{q,r}$)     ▷ Alg. 2
        if $S_{i,j}^{cfg} \neq \emptyset$ then
            Execute $J_{i,j}$ with $S_{i,j}^{cfg}$; remove $J_{i,j}$ from ready queue
        end if
    end if
    if $S_{avail} = \emptyset$ or $S_{cfg} = \emptyset$ then     ▷ $J_{i,j}$ not executed
        Repeat the procedure for the next jobs in the ready queue
    until $S_{avail} = \emptyset$ or every job has been visited
    end if
end procedure
```

The WCET profile, power consumption profile and $m^{opt}$ for each task are also stored in the server.

During the runtime phase, the server keeps track of the status of SMs, which are updated at the release and completion of every job. The server also maintains two containers: a ready queue to keep track of ready jobs and a run queue for currently executing jobs. We sort jobs in the ready queue based on their deadlines, but other policies can also be used, e.g., FCFS or criticality if exists. A set $S_{avail}$ keeps the SMs that are not being used by any job. Whenever a new job arrives, the server invokes the scheduler (explained below) to let it decide whether the server should offload the job to one of the workers right away or leave it in the ready queue. The scheduler is also invoked when a currently running job completes. The worker on which the job was executing notifies the server via the global data structure and the server marks the worker thread as "vacant". Then the SMs that were used by the job are returned back to $S_{avail}$ and the scheduler is invoked to make a scheduling decision for ready jobs.

Based on this framework design, below we present our runtime scheduling algorithms.

*1) Runtime Scheduler:* The scheduler of sBEET is given in Alg. 1. It is invoked by the server when a new job arrives or a current job finishes. The scheduler determines up to two jobs to execute simultaneously on different sets of SMs to avoid the unpredictable delay that can be caused when the two CUDA kernels compete for the same set of SMs. When a new job arrives, it is pushed into the ready queue, and the first job $J_{i,j}$ in the queue is passed to the scheduler. The scheduler first checks the currently available SM set $S_{avail}$. If the GPU is idling ($|S_{avail}| = M$, where $M$ is the total number of SMs on the GPU), it calls the SM allocation algorithm (Alg. 2) to obtain the SM allocation $S_{i,j}^{cfg}$ for $J_{i,j}$. If the GPU is partially utilized ($0 < |S_{avail}| < M$), the scheduler passes the new job $J_{i,j}$ along with the currently-running job $J_{q,r}$ to the SM allocation algorithm so that it can give $S_{i,j}^{cfg}$ to $J_{i,j}$ based on

**Algorithm 2** SM Allocation

```
1: function ALLOCATION($J_{i,j}$, $J_{q,r}$)
2:     $t_{now} \leftarrow$ current time
3:     if $J_{q,r}$ is $nullptr$ then                          ▷ $\Longrightarrow$ GPU is idling
4:         for $m \leftarrow M$ to 1 do
5:             $m' \leftarrow \min(m, m_i^{opt})$
6:             $Q_{i,j}^w \leftarrow \{J_{k,p} \mid \forall p, (\tau_k \neq \tau_q) \wedge (r_{k,p} < f_{i,j}(m'))\}$
7:             SCHEDGEN($J_{i,j}$, $J_{q,r}$, $m'$, $[t_{now}, f_{i,j}(m')]$, $Q_{i,j}^w$)
8:             Compute $E_{pred} = E(t_{now}, f_{i,j}(m'))$ by Eq. (5)
9:         end for
10:        if no generated schedule is feasible then
11:            Choose the schedule with the minimum $E_{pred}$
12:        else
13:            Choose the feasible schedule with the min. $E_{pred}$
14:        end if
15:        return $S_{i,j}^{cfg}$ ▷ the corresponding SM allocation for $J_{i,j}$
16:    else                                      ▷ the GPU is partially occupied
17:        $m' \leftarrow \min(|S_{avail}|, m_i^{opt})$
18:        if $f_{i,j}(m') > f_{q,r} + G_{i,j}^e(M)$ then
19:            return $\emptyset$       ▷ Do not run $J_{i,j}$ in parallel with $J_{q,r}$
20:        else
21:            $Q_{i,j}^w \leftarrow \{J_{k,p} \mid \forall p, (\tau_k \neq \tau_q) \wedge (r_{k,p} < f_{i,j}(m'))\}$
22:            SCHEDGEN($J_{i,j}$, $J_{q,r}$, $m'$, $[t_{now}, f_{i,j}(m')]$, $Q_{i,j}^w$)
23:            if the generated schedule is not feasible then
24:                return $\emptyset$
25:            else
26:                return $S_{i,j}^{cfg}$              ▷ the corresp. SM allocation
27:            end if
28:        end if
29:    end if
30: end function
```

**Algorithm 3** Schedule Generation

```
1: function SCHEDGEN($J_{i,j}$, $J_{q,r}$, $m'$, $[t_{now}, t_{fin}]$, $Q_{i,j}^w$)
2:     /* Generate a schedule for $[t_{now}, t_{fin}]$ */
3:     Place $J_{i,j}$ with $m'$ SMs at $t_{now}$
4:     if $J_{q,r} = nullptr$ then
5:         $t_{next} \leftarrow t_{now}$   ▷ Start time for the next job $J_{k,p} \in Q_{i,j}^w$
6:     else
7:         Place $J_{q,r}$ from $t_{now}$ to $f_{q,r}$
8:         $t_{next} \leftarrow \min(f_{i,j}(m'), f_{q,r})$  ▷ $J_{i,j}$ or $J_{q,r}$'s finish time
9:     end if
10:    /* Consider other jobs in $Q_{i,j}^w$ */
11:    $S_{rem} \leftarrow$ # of remaining (unused) SMs at $t_{k,p}$
12:    for $J_{k,p} \in Q_{i,j}^w$ in ascending order of arrival time do
13:        $m'' \leftarrow \min(S_{rem}, m_k^{opt})$
14:        if $m'' = 0$ then
15:            continue          ▷ Ignore this job from parallel exec.
16:        end if
17:        Place $J_{k,p}$ with $m''$ SMs at $t_{next}$
18:        $t_{next} \leftarrow f_{k,p}(m'')$
19:        if $t_{next} \geq t_{fin}$ then
20:            break                           ▷ Stop schedule generation
21:        end if
22:    end for                                 ▷ Schedule generation done
23: end function
```

the information of $J_{q,r}$. If no valid SM allocation is found ($S_{i,j}^{cfg} = \emptyset$) or all SMs are busy ($S_{avail} = \emptyset$), the scheduler puts $J_{i,j}$ in the ready queue and iteratively checks the next job in the ready queue with the same procedure.

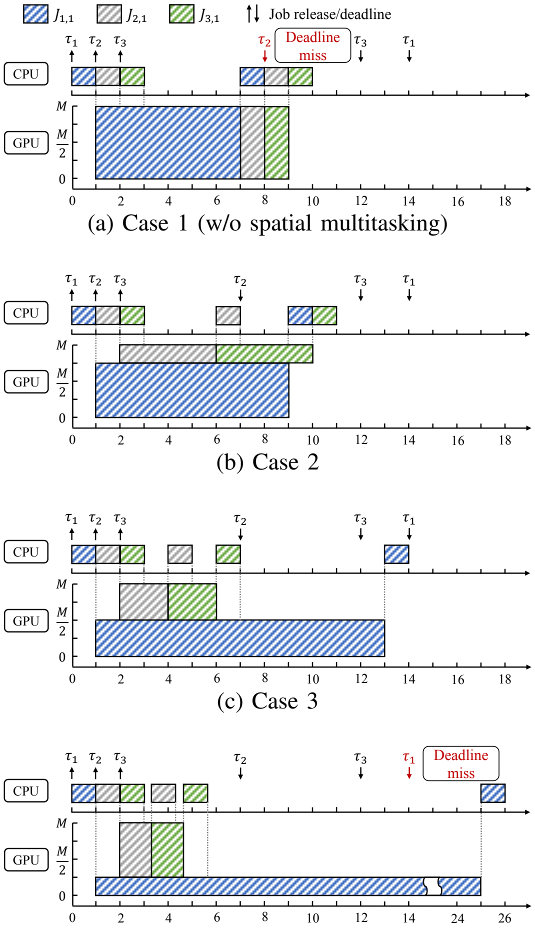
As discussed in Section IV-A, the energy consumption may increase when spatial multitasking is used. To make a trade-off between schedulability and energy efficiency, our SM allocation algorithm adopts the following heuristic strategy:

**SM Allocation.** The proposed SM allocation algorithm uses a job set $Q_{i,j}^w$ for a given job $J_{i,j} \in \tau_i$ to check all the jobs of other tasks that are expected to arrive during $J_{i,j}$'s execution. Formally, $Q_{i,j}^w := \{J_{k,p} \mid \forall p, (J_{k,p} \in \tau_k) \wedge (\tau_k \neq \tau_i) \wedge (r_{k,p} < f_{i,j}(m'))\}$, where $f_{i,j}(m')$ is the expected finish time of $J_{i,j}$ if it begins execution at the current time $t_{now}$ with $m'$ dedicated SMs. The algorithm considers a possible schedule of $J_{i,j} \cup Q_{i,j}^w$ for each $m'$, and chooses the one that leads to the minimum predicted energy consumption in an interval of $[t_{now}, f_{i,j}(m')]$. Note that the decision made by the algorithm for $J_{i,j}$ does not affect the currently running job $J_{q,r}$ since it does not assign SMs that are not in $S_{avail}$.

Alg. 2 depicts the pseudocode of our SM allocation. It takes the jobs passed by Alg. 1 ($J_{i,j}$: the job that needs to be executed, $J_{q,r}$: the currently running job), and returns an SM allocation $S_{i,j}^{cfg}$ for $J_{i,j}$. The detailed steps depend on whether the GPU is idling or not:

- (Alg. 2 line 3 to 15) If the GPU is idling, the algorithm iterates through $m$ from $M$ to 1. For each $m$, it assigns $m' = \min(m, m_i^{opt})$ SMs to $J_{i,j}$, and checks if there is any job that is expected to arrive before $f_{i,j}(m')$, and adds such jobs into $Q_{i,j}^w$ in an ascending order of arriving time. Then the algorithm calls the SchedGen function in Alg. 3 (explained below) to generate a schedule of $J_{i,j} \cup Q_{i,j}^w$ for a time interval $[t_{now}, f_{i,j}(m')]$. The algorithm predicts the energy consumption $E_{pred}$ of the generated schedule by Eq. 5 (line 5 to 8). After this iteration, if none of the generated schedule is feasible, the algorithm chooses the one with the minimum energy consumption. Otherwise, the algorithm chooses the most energy-efficient feasible schedule (line 10 to 14). Finally, the algorithm returns the corresponding SM configuration $S_{i,j}^{cfg}$ for $J_{i,j}$ (line 15).
- (Alg. 2 line 16 to 29) If the GPU is partially occupied ($J_{q,r} \neq nullptr$), the scheduler decides whether $J_{i,j}$ should be dispatched to the worker thread right away. This can be done by comparing the expected finish time of $J_{i,j}$ in two cases: (1) executing $J_{i,j}$ with $m' = \min(|S_{avail}|, m_i^{opt})$ SMs at $t_{now}$ (i.e., $f_{i,j}(m')$), and (2) waiting until $J_{q,r}$ completes and then executing $J_{i,j}$ with all $M$ SMs (i.e., $f_{q,r} + G_{i,j}^e(M)$). If case 2 finishes earlier, the algorithm returns $\emptyset$ (line 19) so that $J_{i,j}$ is put back to the ready queue. This is because in this case, executing $J_{i,j}$ with $m'$ SMs not only takes longer but also likely causes more SMs left idle later. Otherwise, following the same approach as when the GPU is idling, the algorithm calls SchedGen and returns $S_{i,j}^{cfg}$ when the generated schedule is feasible (line 20 to 28).

**Schedule Generation.** Alg. 3 generates a schedule for $J_{i,j} \cup Q_{i,j}^w$ for a given time interval $[t_{now}, t_{fin}]$. The way it generates a schedule is straightforward given that: the server can execute only up to 2 kernels at a time, $J_{i,j}$ starts at $t_{now}$, and the time interval finishes when $J_{i,j}$ completes execution ($t_{fin} = f_{i,j}(m')$). Hence, at first, the function places $J_{i,j}$ (and $J_{q,r}$ if



(a) Case 1 (w/o spatial multitasking)

(b) Case 2

(c) Case 3

(d) Case 4

Figure 3: Scheduling results in Example 3

exists) in the schedule (line 3 to 9). Then, at the time $t_{next}$ when one of the jobs finishes execution, it places $J_{k,p} \in Q_{i,j}^{w}$ in their arrival order by using remaining SMs ($S_{rem}$), until the schedule reaches $t_{fin}$ (line 11 to 22). Note that if there is no remaining SM left, $J_{k,p}$ will be ignored from the schedule generation, assuming it can be executed later (line 14 to 16).

In the following example, we illustrate how the scheduler works at runtime.

**Example 2.** Consider a taskset $\Gamma$ of three *linear-speedup* tasks and a GPU with $M$ SMs. The memory copy operation and GPU execution time of the tasks are listed in Table II. Fig. 3a shows the schedule without spatial multitasking. Under any work-conserving scheduling policies such as FCFS and RM, at $t = 0$, $J_{1,1}$ is scheduled since none of jobs of other tasks are ready. After memory copy, it occupies all the SMs in [1, 7). The following job $J_{2,1}$ is released at $t = 1$, but because of the blocking by $J_{1,1}$, $J_{2,1}$ cannot execute until $t = 7$ and misses the deadline.

The proposed scheduler considers possible future schedules resulted by the SM allocation to the job of interest until this job finishes execution. When $J_{1,1}$ arrives, the scheduler first considers the schedule in Fig. 3a during an interval of $G_{1,1}^{e}(M)$. The scheduler can find that $J_{2,1}$ and $J_{3,1}$ will arrive during this interval based on the information of their periods and offsets, and $J_{2,1}$ will miss the deadline due to the blocking from $J_{1,1}$. Next, the scheduler considers the schedule shown in Fig. 3b where $\frac{3\cdot M}{4}$ SMs is assigned to $J_{1,1}$. In this case, since fewer SMs are given to $J_{1,1}$, an interval of $G_{1,1}^{e}(\frac{3\cdot M}{4})$ is considered, and all of the three jobs are expected to meet their deadlines. The same procedure is conducted with other SM allocations to $J_{1,1}$, e.g., Fig. 3c and Fig. 3d. The schedule in Fig. 3d executes $J_{1,1}$ on just a single SM and results in a deadline miss. After those schedules are generated, the scheduler predicts the energy consumption of each schedule that does not miss any deadline during the interval of interest. In this scenario, all the jobs can meet the deadline in both Fig. 3b and 3c. Hence, the scheduler selects the schedule in Fig. 3b since it has a lower energy consumption computed by the approach in Theorem 1.

Table II: Taskset in Example 2

| Task | $D_i$ | $G_i^e(M)$ | $G_i^{hd}$ | $G_i^{dh}$ | Offset |
|---|---|---|---|---|---|
| $\tau_1$ | 14 | 6 | 1 | 1 | 0 |
| $\tau_2$ | 7 | 1 | 1 | 1 | 1 |
| $\tau_3$ | 10 | 1 | 1 | 1 | 2 |

*2) Time Complexity Analysis:* Here we discuss the time complexity of sBEET. Suppose we have $n$ tasks in the taskset, and at most $K$ jobs can be released by each task during an interval considered by the SM allocation algorithm. The number of jobs considered for each schedule generation (line 12 of Alg. 3) is upper-bounded by $nK$. The procedure to check deadline miss and compute the energy consumption for each schedule is also upper-bounded by $nK$. The number of generated schedules is a constant ($= M$, line 4 of Alg. 2) because it depends on the total number of SMs on the target GPU. So it takes a constant time to select the schedule with the best energy consumption. In order to maintain the ready queue in the server, it takes $O(nK \cdot log(nK))$ to sort the ready jobs in an order of their deadlines. Therefore, the whole procedure of Alg. 1, 2 and 3 takes $O(nK) + O(nK \cdot log(nK)) = O(nK \cdot log(nK))$. If $K$ can be bound to a constant, which is reasonable since the size of $K$ is constrained by task utilization and job's WCET, the time complexity can be expressed as $O(n \cdot log(n))$.

*3) Offline Schedule Generation:* With the algorithms given in Alg. 1 and Alg. 2, we can also generate an offline schedule to statically analyze the schedulability of a given taskset. The offline schedule can be generated for one hyperperiod of the given taskset by simulating job arrivals and their SM allocations using the proposed runtime scheduler. Then, the occurrence of deadline misses can be easily detected from the generated schedule. In order to preserve the execution order of jobs found in the offline schedule at runtime, the jobs should be executed in a non-work-conserving manner; hence, even if the previous job finishes earlier than its expected finish time based on the WCET, the next job should begin execution according to its start time recorded in the offline schedule. For sporadic tasks, we consider the minimum inter-arrival time as periods. However, unlike the runtime scheduler, the offline schedule is generated based on the WCETs of tasks, and can be less energy-efficient due to possible idle times that are only observable at runtime.

## V. Evaluation

In this section, we first present the profiling results of WCET and power consumption, and evaluate the accuracy of our power model. We then check the runtime overhead of sBEET implemented on Xavier AGX. Finally, we conduct experiments to evaluate the effectiveness of sBEET on schedulability and energy consumption compared against existing approaches.

### A. Experiment Setup

The experiments are done on a Jetson AGX Xavier Developer Kit using CUDA 10.0 SDK, running on Ubuntu 18.04. GPU power consumption is measured using the built-in power sensor at the GPU power supply rail every 1 ms, and the energy consumption is estimated by integrating the power consumption records over the duration of GPU taskset execution. To minimize measurement inaccuracy, we fixed the GPU clock frequency to 670 MHz and enabled all CPU cores.

Since the Xavier platform features shared memory between the CPU and the GPU, we ignored the energy consumption during data transfer between the host and the device and limited our focus to processing elements. According to the temperature reported by the built-in sensor during profiling, the observed change in temperature is insignificant during kernel execution on this low-power platform; hence, the potential impact of chip temperature on power consumption is not considered in this work.

**Obtaining Power Parameters.** The power parameters $P^s$, $P^d$ and $P^{idle}$ are obtained using the average of 10-minute measurements from the built-in power sensor under different conditions. $P^s$ was directly measured from the sensor when the GPU is completely idle, i.e., no active SM at all. For $P^d(m)$, $P^d(m = M)$ was first obtained by $P^d(M) = P - P^s$ when all SMs are utilized. For $P^d(m < M)$, Eq. 3 holds [16]; therefore it was estimated by $P^d(m) = \frac{(P-P^s)*m}{M}$. Lastly, with $P^s$ and $P^d(m)$ for each application, we could get $P^{idle}$ for different numbers of SMs by Eq. 2.

**Benchmarks Selection.** In the evaluation, We consider eight different GPU benchmarks whose execution time is not too short ($> 100\mu s$) so that the overlapped kernel execution and its power consumption can be observed: `mmul`, `stereodisparity`, `dxtc` are selected from Nvidia CUDA 10.0 sample programs [3], `hotspot`, `pathfinder`, `bfs` (two benchmarks with different input size) are from the Rodinia GPU benchmark suite [13], and one synthetic computing-intensive kernel which performs vector norm in double precision. CUDA streams are used for asynchronous data transfer and concurrent kernel execution.

**SM Allocation.** The SM allocation is implemented by using *persistent threads*, as done in other previous works [14, 20, 31] on spatial multitasking GPUs. Specifically, when a job is released, the scheduler decides the target SMs that should be assigned to the job. If a thread block is launched on a non-target SM, the thread block stops execution immediately so that non-target (unassigned) SMs can idle without spinning and be ready for other kernels. On the other hand, the thread blocks on the target SMs become persistent; they keep running on the target SMs for the whole lifetime of the kernel in order to finish the work that should have been done by the stopped thread blocks. We use the default number of threads per thread block given by the CUDA code of the benchmarks.

### B. WCET and Power Consumption Profiling

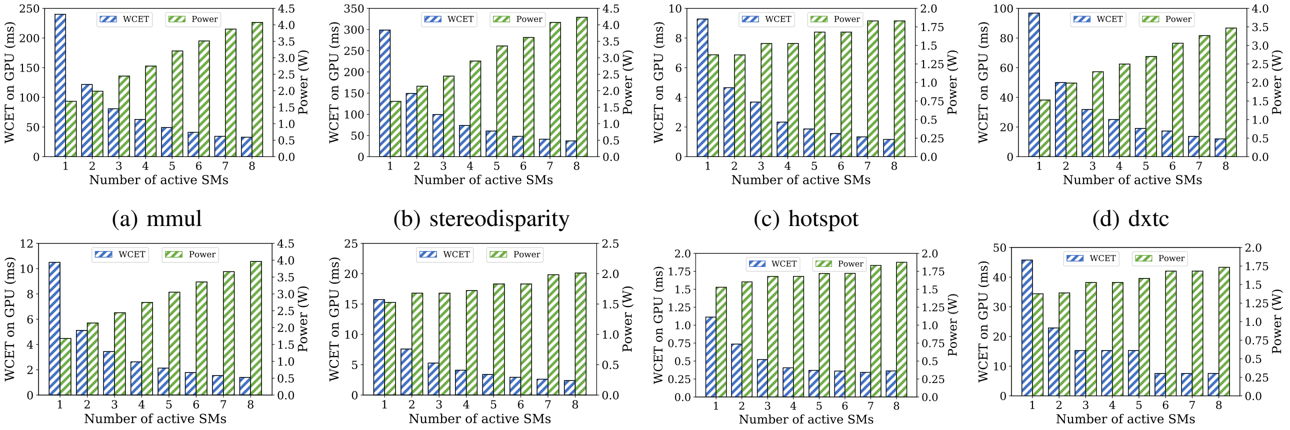
To explore how the number of active SMs affects the GPU power consumption, we conduct experiments using the aforementioned benchmarks. We first profile the cumulative WCET of GPU segments of each benchmark with a different number of SMs since the execution time is directly related to the energy consumption. We then profile the power consumption of benchmarks by taking the average of the measured power by executing each benchmark continuously for 10 minutes. We consider the maximum observed execution time as the WCET. Fig. 4 shows an increase in power consumption and a decrease in WCET as the number of active SMs increases. Three observations can be made here: (i) the WCET of `mmul`, `stereodisparity`, `hotspot`, `dxtc`, `pathfinder` and `bfs_large` is inversely proportional to the number of SMs assigned to it, thereby following the *linear-speedup* model, (ii) for `bfs_small` and the synthetic kernel, there exists $m$ that assigning more than $m$ SMs does not benefit execution time, following the *nonlinear-speedup* model, and (iii) the power consumption increases sublinearly with the number of SMs.

### C. Energy Consumption in an Observation Window

To find out $m_i^{opt}$ in Def. 1, we have observed the energy consumption of each benchmark with a different number of SMs, as shown in Fig. 5. For each benchmark, we choose an observation window which is slightly larger than the WCET of the benchmark when only 1 SM is assigned. As we previously mentioned, the sampling rate of the built-in power sensor is relatively low, possibly causing inaccurate readings when the measuring interval is short. Thus we compute the energy consumption of each schedule during this time interval with different number of SMs by Eq. 5 using the profiling results in Section V-B. For `mmul`, `stereodisparity`, `hotspot`, `dxtc`, `pathfinder` and `bfs_large`, which are *linear-speedup* tasks, the minimum energy consumption is observed when all 8 SMs are assigned, i.e., $m_k^{opt} = M$. Whereas for `bfs_small` and the synthetic kernel, $m_k^{opt} < M$ leads to the best energy consumption.

### D. Prediction of Power Consumption

We evaluate the accuracy of the power model in this section. Since sBEET allows at most two jobs to run on the GPU simultaneously, we create pairs of benchmarks and consider all possible combinations of SM allocations for each pair on the target platform, i.e., $m_1$ idle SMs, $m_2$ SMs for the first



(a) mmul (b) stereodisparity (c) hotspot (d) dxtc

(e) pathfinder (f) bfs_large (g) bfs_small (h) synthetic kernel

Figure 4: Profiling results of WCET and power consumption

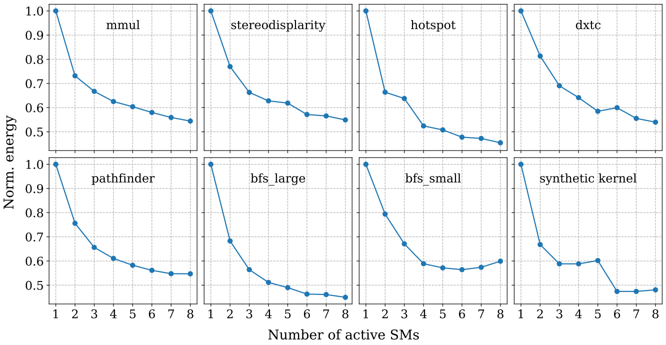


Figure 5: Normalized energy consumption in time window

benchmark of the pair, and $m_3$ for the second one such that $m_1+m_2+m_3=8$. We then measure the power consumption from the built-in sensor and compare it against the predicted value by our power model. Fig. 6 depicts the variance of the error between the measured (observed) and predicted power consumption for each pair of benchmarks. The arithmetic mean of error in power prediction is 5.93% and the average R-squared value (coefficient of determination) of correlation between the measured and predicted power is shown to be 0.87. Since the internal power sensor is used to collect the power consumption, and it has a relatively low sampling rate, which may cause inaccurate readings [12] for the GPU kernels with short duration such as `hotspot`, `pathfinder` and `bfs`, thus resulting in relatively larger modeling error. However, this is not related to the soundness of our power model, as evidenced by the results of the other kernels.

### E. System Evaluation

In this section, we compare the performance of sBEET against the following three approaches: (i) FCFS - the default FCFS scheduling policy of NVIDIA GPU without spatial multitasking, (ii) RM - Rate-Monotonic scheduling of GPU tasks without spatial multitasking (RM), and (iii) STGM - the latest GPU scheduling algorithm proposed in [31] that uses both spatial and temporal multitasking. Under both FCFS and RM, each task uses all eight SMs of the GPU. Under STGM, the SM allocation for each task is statically determined by its offline algorithm. In order to compare the runtime performance of STGM in diverse scenarios, we replaced the pessimistic response-time-based schedulability test of STGM's SM allocation algorithm with a simple version that only checks whether the total utilization of the given taskset exceeds 1.0 so that more tasksets are admitted to run. When $U>1.0$, STGM falls back to RM because STGM cannot find SM allocation for such a taskset. We consider a unified experimental setup to evaluate the runtime performance of various scheduling approaches, with the deadline miss ratio and the energy consumption as evaluation metrics.

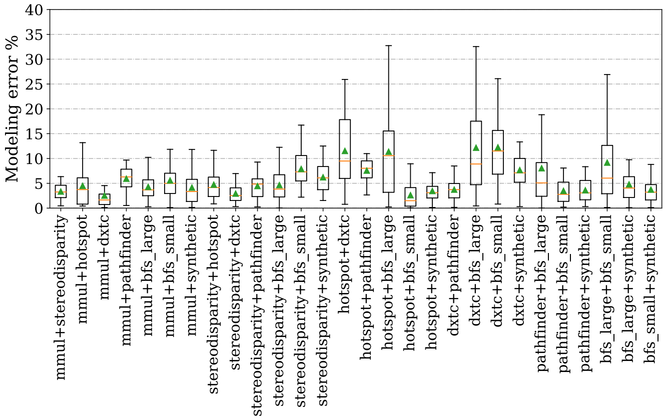
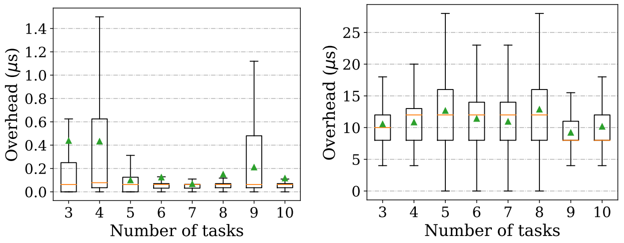
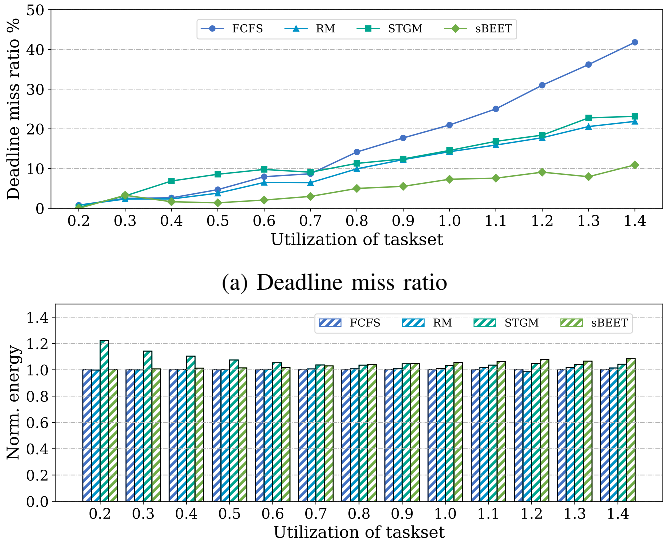


Figure 6: Error of predicted GPU power consumption



(a) Overhead of Alg. 1 (b) Overhead of Alg. 2 and 3

Figure 7: Runtime overhead w.r.t number of tasks



(a) Deadline miss ratio

(b) Overall energy consumption

Figure 8: Runtime results w.r.t. the utilization of taskset

*1) Overhead Measurement:* We measured the runtime overhead of sBEET and the results are shown in Fig. 7. Note that the measured overhead of the scheduler (Alg. 1) excludes the overhead of SM allocation (Alg. 2) and SchedGen (Alg. 3). The same experiment setups are used here as stated in Section V-A. To obtain the overheads of the proposed runtime algorithms in Section IV-B , we randomly generated tasksets consisting of various number of tasks from the benchmark pool. The total utilization of each taskset is set to be 1.0, and the running duration is set to 10 minutes. Since the proposed scheduler makes an SM allocation decision at each job arrival, the increase of the number of tasks does not necessarily leads to the increase of overheads. Since the maximum total overhead of the algorithms is much less than 100 $\mu$s, we conclude that the sBEET framework is suitable to use on embedded platforms at runtime.

*2) Effect of Taskset Utilization:* We generate 1,000 random tasksets for each experiment and execute them on real hardware. The following parameters are considered for each task generation: workload type (one of the six benchmarks mentioned before), task utilization (defined in Sec. II-B and determined by the UUniFast algorithm [11]), and the initial release offset ($[0, \frac{T_i}{2}]$). Once the workload type is chosen randomly among the benchmarks, the WCET of the task is determined automatically from the profiles, and the period (equal to deadline, i.e., $T_i = D_i$) is obtained by dividing its WCET by utilization. For each generated taskset, we run FCFS, RM, STGM, and sBEET each for 10 seconds (we did not observe a meaningful difference in deadline miss ratios even if the scheduler runs for a longer time). For FCFS and RM, the tasks run on a single stream with only one worker since they do not use spatial multitasking, and it represents the synchronization-based real-time GPU access approach [15, 24, 30]. For STGM, eight workers are created since STGM does not limit the number of jobs that can run simultaneously on the GPU.

In Fig. 8, we show the performance of the four scheduling approaches for various utilization settings. Fig. 8a presents the deadline miss ratio under the four approaches, and sBEET has the lowest deadline miss ratio among them. Note that, when $U \leq 0.7$, the performance of STGM is the worst among the four approaches. This happens because the blocking time by a GPU kernel under STGM is likely to be longer than that under RM when the SMs are not fully utilized. When $U \geq 0.8$, FCFS becomes the worst among the four approaches. The curves of STGM and RM overlap due to the fallback mentioned in Section V-E.

Fig. 8b shows the runtime energy consumption of the four approaches, normalized to the case of FCFS. We observe that, when $U \leq 0.7$, STGM has the biggest overall energy consumption because it first assigns the minimum possible number of SMs to each task and incrementally increases the number only for those showing a large reduction in task utilization. As $U$ gets larger, the energy consumption of FCFS, RM, and STGM becomes slightly lower than our proposed scheduler because ours use spatial multitasking to achieve better schedulability, the use of which inevitably increases energy consumption as stated in Section IV-A. Another reason is that FCFS, RM and STGM have higher deadline miss ratios, as can be seen in Fig. 8a. In other words, during the same observation interval, they have fewer completed jobs compared to sBEET.

*3) Effect of Heavy/Light Task Ratios:* In order to better understand the schedulability characteristics of sBEET for light tasks that can suffer from long blocking time caused by heavy tasks released earlier [25], we conduct experiments using randomly-generated bi-modal tasksets. Based on the WCET profiles of each benchmark, we consider `hotspot`, `pathfinder`, `bfs` and the synthetic kernel as light tasks, and `mmul`, `stereodisparity`, and `dxtc` as heavy tasks. We keep the same total utilization of $U = 0.9$ for each taskset. The light and heavy tasks are generated according to the ratio of the heavy tasks until the total utilization exceeds the target utilization. The utilization of each light task is randomly selected between [0.2, 0.4] and heavy tasks between [0.05, 0.2]. Fig. 9 demonstrates the runtime deadline miss ratio of light tasks as the percentage of heavy tasks increases in a taskset. Since sBEET takes into account tasks' possible future arrivals to find the right number of SMs and decide when to launch the jobs in a non-work-conserving manner, long blocking time from other jobs can be minimized, as shown in Fig. 2. Therefore, sBEET has a better performance in meeting the deadlines of light tasks than the other approaches.

*4) Effect of Spatial Multitasking:* Finally, we conduct additional experiments to compare the energy consumption of STGM and sBEET since they both use spatial multitasking. We use the tasksets that are said schedulable by the STGM's offline schedulability test (Eq. 9 in [31]), which guarantees no deadline miss at runtime. We randomly select tasks from our benchmark pool and choose periods within a range of [100, 500] ms to generate each taskset with a fixed number of tasks.

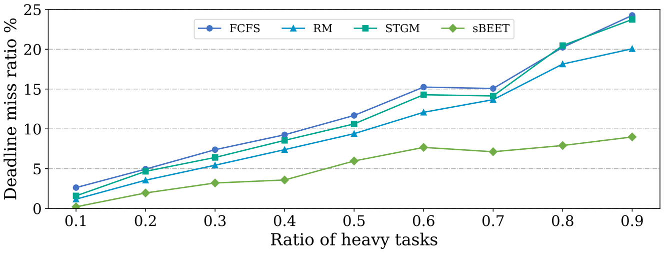


Figure 9: Runtime deadline miss ratio of light tasks w.r.t. ratio of heavy tasks

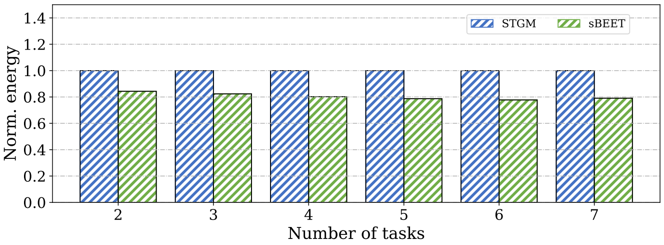


Figure 10: Comparison of runtime energy consumption of STGM and the proposed work

The measurement results of runtime energy consumption are shown in Fig. 10. Compared to STGM that does not limit the number of workers to two, sBEET can save 15% to 21% of energy in actual measurement while also having a 0% deadline miss ratio. These results are consistent with the analysis in Section IV-A, and also supports the reasoning that having more workers may not help improve energy consumption and schedulability.

### *F. Discussion*

While experimental results have demonstrated the benefit and effectiveness of our scheduler, there are some limitations that we would like to discuss. At first, co-scheduled kernels may experience additional timing interference due to contention on shared memory resources of the GPU, which our work does not take into account. Although we did not observe any discernible slowdown of co-scheduled kernels in our experimental setup, probably due to a relatively small number of SMs and the high memory bandwidth of Xavier AGX (58.4 GB/s), the negative impact of memory interference can be a serious problem on GPUs with a large number of SMs or low memory bandwidth. However, our work can be co-used with GPU cache and DRAM partitioning methods [19], which can significantly reduce memory interference and achieve better performance isolation.

Another limitation is that our work considers only the energy consumption of the GPU, although GPU kernel execution draws power from CPUs and other hardware components for data copy and miscellaneous operations. It will be more challenging to optimize the energy consumption of the whole hardware platform including GPU, CPU, memory, etc. We leave this as part of future work.

## VI. Conclusion

In this paper, we first presented the analysis of GPU energy consumption in the presence of spatial multitasking which allows simultaneous execution of multiple GPU kernels. Our analysis suggests that, although spatial multitasking benefits schedulability, its use can lead to energy inefficiency due to the power consumption of idling SMs. Based on this analysis, we then proposed sBEET, a runtime scheduler that balances energy efficiency and real-time performance by utilizing spatial multitasking and predicting the resulting energy consumption. Experimental results using our implementation on real hardware indicate that sBEET reduces deadline misses significantly compared to the other approaches while consuming energy similar to the non-spatial multitasking methods, and achieves better energy efficiency than the others for tasks that satisfy their deadlines.

As GPUs are increasingly required in cyber-physical systems, the high energy consumption of GPUs is becoming an important issue. Our results can serve as a basis to extend the energy-efficient scheduling approach to more powerful, high-end GPUs on which the performance trend of the workloads might be different, and we also plan to consider heterogeneous multi-GPU systems in the future.

## Acknowledgment

This work is supported by the National Science Foundation (NSF) grants 1943265 and 1955650.

## References

[1] Jetson AGX Xavier developer kit. https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[2] Jetson AGX Xavier thermal design guide. https://static5.arrow.com/pdfs/2018/12/12/12/22/1/565659/nvda_/manual/jetson_agx_xavier_thermal_design_guide_v1.0.pdf.

[3] Nvidia CUDA samples. https://github.com/NVIDIA/cuda-samples.

[4] Nvidia multi-process service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[5] Volta tuning guide. https://docs.nvidia.com/cuda/volta-tuning-guide/index.html.

[6] M. Abdel-Majeed, D. Wong, and M. Annavaram. Warped gates: Gating aware scheduling and power gating for GPGPUs. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 111–122. IEEE, 2013.

[7] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for GPGPU spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12, 2012.

[8] P. Aguilera, K. Morrow, and N. S. Kim. QoS-aware dynamic resource allocation for spatial-multitasking GPUs. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 726–731, 2014.

[9] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. GPU scheduling on the NVIDIA TX2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, 2017.

[10] C. Basaran and K. Kang. Supporting preemptive task executions and memory copies in GPGPUs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 287–296, 2012.

[11] E. Bini. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 05 2005.

[12] R. A. Bridges, N. Imam, and T. M. Mintz. Understanding gpu power. a survey of profiling, modeling, and simulation methods. *ACM Computing Surveys*, 49(3), 9 2016.

[13] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[14] G. Chen, Y. Zhao, X. Shen, and H. Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of GPU. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

[15] G. Elliott and J. Anderson. Globally scheduled real-time multiprocessor systems with GPUs. *Real-Time Systems*, 48:34–74, 05 2012.

[16] S. Hong and H. Kim. An integrated GPU power and performance model. *ACM SIGARCH Computer Architecture News*, 38:280, 2010.

[17] S. Hosseinimotlagh and H. Kim. Thermal-aware servers for real-time tasks on multi-core GPU-integrated embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 254–266. IEEE, 2019.

[18] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: methodology and empirical data. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 93–104, 2003.

[19] S. Jain, I. Baek, S. Wang, and R. Rajkumar. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 29–41, 2019.

[20] J. Janzén, D. Black-Schaffer, and A. Hugo. Partitioning gpus for improved scalability. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 42–49, 2016.

[21] Y. Kang, W. Joo, S. Lee, and D. Shin. Priority-driven spatial resource sharing scheduling for embedded graphics processing units. *Journal of Systems Architecture*, 76:17–27, 2017.

[22] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. RGEM: A responsive GPGPU execution model for runtime engines. In *2011 IEEE 32nd Real-Time Systems Symposium*, pages 57–66, 2011.

[23] S. Kato, C. M. University, T. U. of Tokyo, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, Portland, OR, 2011. USENIX Association.

[24] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar. A server-based approach for predictable GPU access with improved analysis. *Journal of Systems Architecture*, 88:97–109, 2018.

[25] H. Lee and J. Lee. Limited non-preemptive EDF scheduling for a real-time system with symmetry multiprocessors. *Symmetry*, 12:172, 01 2020.

[26] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Kim, T. Aamodt, and V. Janapa Reddi. Gpuwattch: enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, 41, 07 2013.

[27] Y. Liang, H. P. Huynh, K. Rupnow, R. S. M. Goh, and D. Chen. Efficient GPU spatial-temporal multitasking. *IEEE Transactions on Parallel and Distributed Systems*, 26(3):748–760, 2015.

[28] S. Mittal and J. Vetter. A Survey of Methods For Analyzing and Improving GPU Energy Efficiency. *ACM Computing Surveys*, 47, 04 2014.

[29] I. S. Olmedo, N. Capodieci, J. L. Martinez, A. Marongiu, and M. Bertogna. Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 213–225, 2020.

[30] P. Patel, I. Baek, H. Kim, and R. Rajkumar. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[31] S. Saha, Y. Xiang, and H. Kim. Stgm: Spatio-temporal gpu management for real-time tasks. In *2019 IEEE 25th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–6, 2019.

[32] J. Sun, J. Li, Z. Guo, A. Zou, X. Zhang, K. Agrawal, and S. Baruah. Real-time scheduling upon a host-centric acceleration architecture with data offloading. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 56–69, 2020.

[33] Q. Sun, Y. Liu, H. Yang, Z. Luan, and D. Qian. Smqos: Improving utilization and energy efficiency with qos awareness on gpus. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–5, 2019.

[34] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on GPUs. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, 2014.

[35] Z.-G. Tasoulas and I. Anagnostopoulos. Improving GPU performance with a power-aware streaming multiprocessor allocation methodology. *Electronics*, 8(12), 2019.

[36] P.-H. Wang, C.-L. Yang, Y.-M. Chen, and Y.-J. Cheng. Power gating strategies on GPUs. *TACO*, 8:13, 10 2011.

[37] Y. Wang and H. Kim. Work-in-progress: Understanding the effect of kernel scheduling on gpu energy consumption. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 584–587, 2019.

[38] Y. Wang and N. Ranganathan. An instruction-level energy estimation and optimization methodology for GPU. In *2011 IEEE 11th International Conference on Computer and Information Technology*, pages 621–628, 2011.

[39] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Quality of service support for fine-grained sharing on gpus. pages 269–281, 06 2017.

[40] Y. Xiang and H. Kim. Pipelined data-parallel CPU/GPU scheduling for multi-DNN real-time inference. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 392–405. IEEE, 2019.

[41] H. E. Zahaf, A. Benyamina, R. Olejnik, and G. Lipari. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture*, 74, 01 2017.

[42] H. Zhou, G. Tong, and C. Liu. GPES: a preemptive execution system for GPGPU computing. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 87–97, 2015.