# Towards Energy-Efficient Real-Time Scheduling of Heterogeneous Multi-GPU Systems

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

***Abstract*—With the increasing demand for computational power, research on general-purpose graphics processing units (GPUs) has been active for various real-time systems spanning from autonomous vehicles to real-time clouds. While the use of GPUs can significantly benefit compute-intensive tasks with timing constraints, their high power consumption becomes an important problem given that it is not rare to see multiple GPUs in today's systems. In this paper, we present our study towards energy-efficient real-time scheduling in heterogeneous multi-GPU systems. We first make observations using a custom power monitoring setup that, in a multi-GPU system, conventional task allocation approaches for multiprocessors do not lead to energy efficiency and there is no clear winner. Then we propose a multi-GPU real-time scheduling framework, sBEET-mg, that builds upon prior work on single-GPU systems and makes offline and runtime scheduling decisions to execute a given job on the energy-optimal GPU while exploiting spatial multitasking on each GPU for better concurrency and real-time performance. We implemented the proposed framework on a real multi-GPU system and evaluated it with randomly-generated task sets of benchmark programs. We also experimentally simulated our method in a system containing more GPUs. Experimental results show that sBEET-mg reduces deadline misses by up to 23% and 18% compared to the conventional load distribution and load concentration methods, respectively, while simultaneously achieving lower energy consumption than them.**

## I. Introduction

Graphics processing units (GPUs) are attracting much attention due to their outstanding performance over CPUs by allowing huge data parallelism. With the increasing demand driven by data-driven and machine-intelligent applications, research on real-time GPU multitasking becomes more and more popular while leaving their high power consumption as an open problem. According to [1], the high power consumption of GPUs has a significant impact on scalability, reliability and feasibility. An increase in power consumption also raises the risk of thermal violations [2–4]. Without proper management, these issues can be worse in a heterogeneous multi-GPU system which is not rare to see in today's computing environment.

In a multi-GPU system, the workload allocation methods can be generally classified into load distribution and load concentration. For load distribution, due to the fact that CUDA kernels rarely fully utilize all the internal computing units of a GPU [5], the idle energy consumption of the computing units of an active GPU causes energy inefficiency [6, 7] and this issue is likely to be magnified in a multi-GPU system. For load concentration, as we will discuss more details later, different tasks may have different *energy-preferred* GPUs; hence, packing and offloading to the same GPU while keeping other GPUs idle does not necessarily lead to better energy efficiency than load distribution. The problem gets more complicated in real-time systems, since tasks have their own arriving patterns with different timing requirements.

This paper paves a new way to address the energy efficiency and scheduling problem in heterogeneous multi-GPU real-time systems. To gain a precise understanding of power usage characteristics, we analyze a multi-GPU system consisting of two heterogeneous GPUs with a custom hardware tool. With the obtained power characteristics of benchmark programs on different GPUs, we give observations on the energy consumption of a multi-GPU system when different scheduling strategies are applied. Based on these, we present a multi-GPU scheduling framework, sBEET-mg, by extending the latest energy-aware real-time scheduling approach [6] to a multi-GPU system. sBEET-mg allocates tasks to their energy-optimal GPUs offline and performs runtime migration based on the estimation of the resulting energy consumption of all GPUs in the system. It also takes advantage of spatial multitasking to improve real-time performance without losing energy efficiency.

To evaluate the performance of sBEET-mg, the framework is implemented in the multi-GPU system we built. We conduct experiments using randomly-generated tasksets of well-known benchmarks to compare the schedulability and energy consumption of our framework against three existing approaches based on load concentration and load distribution. By judiciously executing jobs on the right GPUs with a proper number of GPU's internal computing units, sBEET-mg achieves lower energy consumption as well as deadline misses. The contribution of this work is summarized as follows:

- We analyze the power usage characteristics of various benchmarks on two recent NVIDIA architectures using precise measurements from our own power monitoring setup. This leads to observations that neither conventional load concentration nor load distribution scheduling strategies are preferable for energy efficiency in a multi-GPU system.
- To the best of our knowledge, the proposed sBEET-mg framework is the first attempt to simultaneously address the timeliness and energy efficiency in a heterogeneous multi-GPU environment. It builds upon the latest work but includes several unique approaches, including offline allocation of tasks to energy-preferred GPUs and runtime

job migration with spatial multitasking and energy consumption estimation across all GPUs in the system.
- We conduct experiments using a real heterogeneous multi-GPU platform as well as simulation of larger scale systems. Experimental results indicate that sBEET-mg can achieve up to 23% and 18% of reduction in deadline misses compared to the conventional load concentration and distribution approaches while consuming less energy than them at the same time.

## II. Related Work

**Real-Time GPU Scheduling.** Real-time scheduling methods for GPU tasks can be categorized into two types: temporal and spatial multitasking. Temporal multitasking views each GPU as an indivisible, minimum unit of resource and focuses on time-sharing of the GPU. Given that many GPUs provide no support or only a limited level of preemption, many earlier studies have modeled a GPU as a non-preemptive resource [8–12]. In particular, Elliott et al. [10] considered a multi-GPU system where a $k$-exclusion locking protocol was used to assign tasks to $k$ GPUs. This allows the system to utilize multiple GPUs in a work-conserving manner, but can result in poor energy consumption as we will show later. In addition, their focus was limited to homogeneous GPUs and no performance variation across GPUs was considered. Spatial multitasking, on the other hand, explicitly takes into account internal processing units of a GPU, such as NVIDIA's Streaming Multiprocessors (SMs) and AMD's Compute Units (CUs),[1] and allows one GPU to execute two or more GPU tasks at the same time by using *persistent threads* [13–15]. Recent studies have shown that spatial multitasking offers better performance isolation and concurrency [16] and better schedulability and resource utilization [17, 18] than spatial multitasking for real-time workloads. However, their focus was primarily on a single GPU and none of them considered energy efficiency along with the timeliness of GPU tasks.

**GPU Energy Efficiency.** Prior work on GPU energy management has mainly focused on regulating the number of active SMs [13, 19–21]. This is based on the assumption that, if the GPU hardware supports SM-level power gating, unused SMs can be turned off and energy consumption can be reduced. For example, Hong and Kim [19] focused on finding the optimal number of SMs for the highest performance-per-Watt. Aguilera et al. [13] and Sun et al. [20] proposed QoS-aware SM allocation techniques based on spatial multitasking to provide both performance and energy efficiency. However, these approaches have been tested using only analytical power models or simulation, and the claimed benefits are difficult to obtain in today's commercial GPUs because even the latest GPU architectures do not support SM-level power gating. So some SMs left unused by those methods can continue to consume active-idle power until all SMs of the GPU become fully idle. The lack of capability to power-gate individual SMs also makes the energy management problem of GPUs different from that of multi-core CPUs.

Recently, Wang et al. [6] proposed an energy-efficient real-time GPU scheduler, called sBEET. They first showed that although spatial multitasking benefits schedulability, it may lead to an energy-inefficient schedule due to the active-idle power consumption of unused SMs. Then they proposed a runtime scheduler that balances the energy inefficiency caused by spatial multitasking with improved real-time performance in a single GPU system. Our work is motivated by this and aims to generalize to a system equipped with multiple heterogeneous GPUs.

## III. Background and System Model

### A. Background

Our description here is based on NVIDIA GPUs and the CUDA programming abstractions but it generally applies to other types of GPUs, e.g., AMD's ROCm platform and HIP runtime APIs. For more information, interested readers can refer to [6, 16, 22–25].

**GPU Execution Model.** GPU programs written in CUDA can make processing requests to a GPU at runtime. The general sequence for running a GPU program is as follows: (i) allocate GPU memory, (ii) copy input data from main memory to GPU memory, (iii) request to launch the GPU program code (called *kernel*), (iv) copy the results back from GPU to main memory, and (v) deallocate GPU memory. While the CUDA memory model by default separates GPU and main memory spaces, it offers a unified memory model that eliminates the need for explicit data copies between GPU memory and main memory.

CUDA provides *streams* as means to control concurrency. All memory copy and kernel execution requests on the same CUDA stream are executed sequentially. However, different CUDA streams can run in an overlapped manner as long as resources are available, thereby allowing better concurrency. Once launched, a kernel is executed by using all available SMs on the GPU. CUDA APIs do not provide an option to determine the number of SMs used by each kernel, but the spatial multitasking technique [16, 17, 26, 27] implements this in software and provides a controlled way to execute multiple kernels in parallel.

**GPU Power Management.** As a GPU consists of multiple SMs, the power management of the GPU happens at both the SM level and the device level. An SM goes to the active-idle state as soon as it is not used. When all of the SMs are unused, the GPU is power-gated[2] and each SM no longer consumes active-idle power. In other words, if only one SM is active, the GPU is not power-gated and the other SMs consume active-idle power. While SM-level power gating has been studied extensively in the literature to achieve better energy efficiency [19, 28], our experiments have confirmed that

[1] We will use SMs to refer to those internal processing units in the rest of the paper.

[2] Since the details of NVIDIA GPU's power management mechanisms are not publicly available, we are unsure if it is actually power-gated or just clock-gated. Nonetheless, we use the term "power gating" since it is generally used in the literature of GPU power management.

it is still not available on NVIDIA's latest Ampere architecture. This matches with the observations of the recent paper [6].

When the GPU is left fully idle for a relatively long time, it enters a deeper low-power mode. This time interval is observed to be approximately 2 seconds in our experiments. While exploiting this power state would be beneficial in interactive systems, we do not consider it in this work since such a long idle time is hard to expect in real-time systems serving periodic or sporadic workloads.

### B. System Model

We describe our models for the hardware platform, tasks, and power and energy consumption. The summary of the notation is listed in Table I.

**Platform Model.** We consider a single-ISA system $\Pi$ consisting of $\omega$ heterogeneous GPUs. The $k$-th GPU in the system is denoted by $\pi_k$, and each GPU is characterized by its power model, computational capacity and clock speed. The GPU $\pi_k$ consists of $M_k$ SMs, each of which is an independent computing unit from the view of spatial multitasking. We use $type(\pi_k)$ to denote the type of the GPU device $\pi_k$, e.g., $type(\pi_k) = type(\pi'_k)$ means two GPUs are identical.

**Task Model.** We consider a taskset $\Gamma$ consisting of $n$ sporadic GPU tasks with fixed priority and 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 $\tau_i$,[3] and we assume that the input size of each job of a task is constant along the time. Following the idea of spatial GPU multitasking [16, 17, 26, 27], each job $J_{i,j}$ of the task $\tau_i$ can execute with a different number of SMs on a different GPU. Hence, we use $G_{i,j}(m, \pi_k)$ to represent the WCET of $J_{i,j}$, where $m$ denotes the number of SMs used by $J_{i,j}$ on the GPU $\pi_k$. $G_{i,j}(m, \pi_k)$ is given by the sum of the following three parameters:

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

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

With the above parameters, a job's finish time can be estimated from the start of the job and we use $f_{i,j}$ to denote it. The *utilization* of a task $\tau_i$ on a GPU $\pi_k$ is defined as the average utilization when different number of SMs are assigned, and it is computed as $U_i(\pi_k) = \frac{\sum_{m=1}^{M_k} U_i(m,\pi_k)}{M_k}$, where $M_k$ is the total number of SMs on the GPU $\pi_k$. The utilization of $\tau_i$ with $m$ SMs on $\pi_k$ is $U_i(m, \pi_k) = \frac{G_i(m,\pi_k)}{T_i}$. The *GPU utilization* $U(\pi_k)$ is the summation of all the tasks that are assigned to the GPU $\pi_k$, i.e., $U(\pi_k) = \sum U_i(\pi_k)$. Without loss of generality, we assume a discrete-time system where timing parameters can be represented in positive integers.

[3]For simplicity, we may omit the subscript $j$ and use $J_i$ when we do not need to distinguish individual jobs.

Table I: Symbols and their definitions in this work

| Notation | Definition |
|---|---|
| $\pi_k$ | The $k$-th GPU in the system |
| $M_k$ | The total number of SMs on the GPU $\pi_k$ |
| $M_k^{limit}$ | The number of SMs that allowed (by the user) on the $\pi_k$ |
| $G_i$ | The cumulative WCET of GPU segments of task $\tau_i$ |
| $T_i$ | The period of task $\tau_i$ |
| $D_i$ | The relative deadline of task $\tau_i$, and $D_i \leq T_i$ |
| $J_{i,j}$ | The $j$-th job of task $\tau_i$ |
| $r_{i,j}$ | The arrival time of $J_{i,j}$ |
| $d_{i,j}$ | The absolute deadline of $J_{i,j}$ |
| $f_{i,j}$ | The estimated finish time of $J_{i,j}$ |
| $m$ | Number of SMs |
| $G_{i,j}$ | The WCET of $J_{i,j}$ |
| $G_i^{hd}(\pi_k)$ | The WCET of device to host memory copy of $\tau_i$ on $\pi_k$ |
| $G_i^{dh}(\pi_k)$ | The WCET of device to host memory copy of $\tau_i$ on $\pi_k$ |
| $G_i^{e}(m, \pi_k)$ | The WCET of kernel execution of $\tau_i$ on $\pi_k$ with $m$ SMs |
| $U_i(\pi_k)$ | The utilization of task $\tau_i$ on $\pi_k$ |
| $U(\pi_k)$ | The utilization of $\pi_k$ |

**Power Model.** Following the power modeling approach in [6, 29, 30], the power consumption of a GPU at time $t$ can be represented as follows:

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

where $P^s$ is the static power consumption, $P^d$ is the dynamic power consumption from active SMs, and $P^{idle}$ is the power consumption from idle SMs. Specifically, $P^d$ is the power consumption required to execute kernels on SMs, and depends on the kernel characteristics including memory access patterns and the number of SMs used [30]. It can be decomposed into a linear sum of per-SM power consumed by each job. For a subset of jobs $\mathrm{J} = \{J_1, J_2, ...\}$ that are executing simultaneously on the GPU $\pi_k$ at time $t$, the power consumption of the GPU $\pi_k$, $P_k$, can be computed as follows:

$$P_k = \begin{cases} P_k^s + \sum\limits_{J_i \in \mathrm{J}} P_{k,i}^d(m_i) + P_k^{idle}(M_k - \sum\limits_{J_i \in \mathrm{J}} m_i) & \text{if } \mathrm{J} = \emptyset \\ P_k^s & \text{if } \mathrm{J} \neq \emptyset \end{cases} \tag{2}$$

where $m_1, m_2, ...$ are the number of SMs that are being used by $J_1, J_2, ...$ at time $t$ ($\sum m_i \leq M_k$). $P_{k,i}^d(m_i)$ is the dynamic power consumption of $J_i$ on $\pi_k$ with $m_i$ active SMs. $P_k^{idle}(m)$ is the idle power consumption of $m$ inactive SMs, and $M_k$, as defined previously, is the total number of SMs on the GPU $\pi_k$. Since dynamic and idle power is known to be linear to the number of SMs [30], $P_{k,i}^d(m_i) = m_i \cdot P_{k,i}^d(1)$ and $P_k^{idle}(m) = m_i \cdot P_k^{idle}(1)$ holds, respectively. Note that when all SMs on the GPU are idle (i.e. $\sum m_i = 0$), the GPU is power-gated and there is no power consumption from $P_k^d$ and $P_k^{idle}$, i.e. $\sum P_k^d(0) = 0$ and $P_k^{idle}(M_k) = 0$.

In this paper, we directly measured these power parameters

of using our test-bed setup (Sec. IV-A), but they can also be estimated using analytical methods [30].

**Energy Consumption.** We adopt the energy computation method in Eq. 5 in [6]. Let us consider a set of jobs $\mathrm{J} = \{J_1, J_2, ...\}$ that are *scheduled* on the GPU $\pi_k$ during a time interval $[t_1, t_2]$. 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 $x_i^m(t)$ that returns 1 if the $m$-th SM is actively used by a job $J_i$ at time $t$, and 0 otherwise. Using this, the energy consumption on a single GPU $\pi_k$ can be computed by:

$$E_k([t_1,t_2]) = \int_{t_1}^{t_2} \Bigg( P_k^s + \sum_{J_i \in \mathrm{J}} \Big( P_{k,i}^d \big( \sum_{m=1}^{M_k} x_i^m(t) \big) \Big) + P_k^{idle} \Big( M_k - \sum_{J_i \in \mathrm{J}} \sum_{m=1}^{M_k} x_i^m(t) \Big) \Bigg) dt \tag{3}$$

And further, the total energy consumption of all GPUs in the the system $\Pi$ can be obtained by:

$$E([t_1,t_2]) = \sum_{\forall \pi_k \in \Pi} E_k([t_1,t_2]) \tag{4}$$

In the above modeling, we did not explicitly consider other on-device components such as copy engines, caches, and buses. However, their power consumption is relatively small compared to that of SMs and can be captured as part of $P_s$ and $P_d$. We will later show with our experiments that our power and energy models are faithful enough to use for making energy-efficient scheduling decisions.

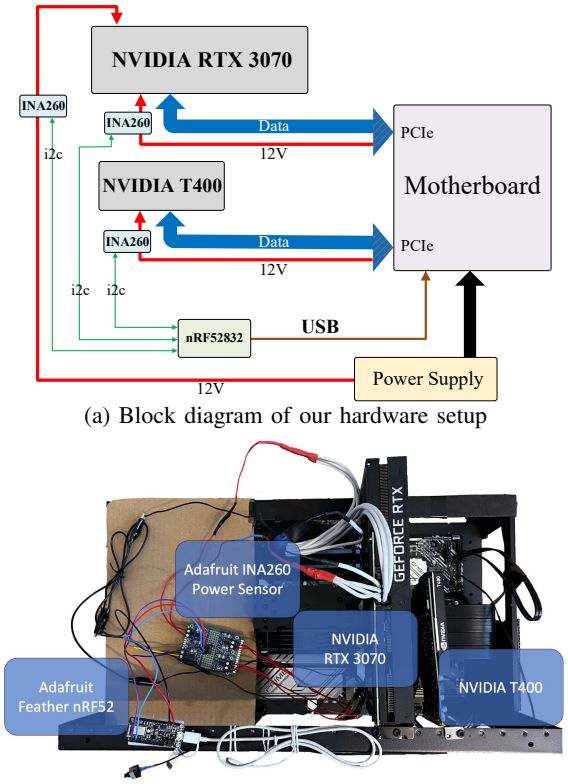
## IV. Energy Usage Characteristics of Multi-GPU Systems

The energy consumption of heterogeneous multi-GPU systems is hard to predict since there is no correlation of dynamic power parameters ($P_d$ and $P_{idle}$) and kernel execution time ($G_i$) across different types of GPUs. In this section, we focus on a system equipped with two GPUs and explore the impact of scheduling policies on energy consumption.

### A. Hardware Setup

The system used in this work consists of one NVIDIA RTX 3070 and one NVIDIA T400. RTX 3070 is based on the latest Ampere architecture. It has 8 GB of global memory and 46 SMs with 5888 CUDA cores. All the SMs share a L2 cache of 4096 KB. Another GPU in our system, T400, is based on the Turing architecture, a predecessor of Ampere. It has 2 GB global memory and 6 SMs with 384 CUDA cores, while 512 KB of L2 cache is shared among all the SMs. For both GPUs, data connection is established directly from the GPU to the PCI Express (PCIe) of the motherboard. During experiments, we fixed the SM clock speed of both GPUs to the maximum, i.e., 1725 MHz for RTX 3070 and 1425 MHz for T400, and both GPUs were able to maintain their frequencies without throttling.

It is worth noting that, although some NVIDIA devices provide power readings via `nvidia-smi` using a built-in




(a) Block diagram of our hardware setup

(b) Implementation of the block diagram

Figure 1: Multi-GPU system with a power monitoring tool

power sensor, its accuracy is not high ("+/- 5 watts" according to the official document [31]) and the power reading is not available on the T400 device. The power measurements by the built-in sensor may show anomalies and need to be corrected as the prior work suggested [32].

Due to these reasons, we developed a custom hardware tool to obtain a precise measurement of power consumption by each GPU. Fig. 1 shows our system with two GPUs connected to the power monitoring tool. We used an INA260 sensor [33] for each of the power supply lines of the GPUs. We used PCIe risers and cut the 12V power lines to install INA260 sensor sensors in series. Due to the high power consumption of NVIDIA RTX 3070 and the limitation of PCIe standard power provision, i.e., 75 Watt, the GPU receives power from both PCIe and the power supply. However, the power provided by PCIe is sufficient for NVIDIA T400 and it does not require any external power supply. We used an nRF52832 SoC [34] to configure the sensors to sample voltage and current. The maximum sampling rate we could obtain from the I2C protocol of the INA260 sensor is 500 Hz, which leads to one sample for every two milliseconds. The data of the sensors are combined and sent to the same computer via USB cable to ensure the best timing synchronization between GPU states and power measurements. Each power sample is recorded in milliWatt, and a high-resolution timestamp is added to each sample as soon as the sample arrives. The power consumption of RTX 3070 is the summation of its power drawn from both PCIe and the power supply. It should be noted that the power

Table II: Power parameters of benchmarks and GPUs

(a) Dynamic power of benchmarks

| Benchmark$_i$ | $P^d_{0,i}(1)$ | $P^d_{1,i}(1)$ |
|---|---|---|
| MatrixMul | 3.77 W | 2.06 W |
| Stereodisparity | 1.63 W | 0.98 W |
| Hotspot | 1.14 W | 0.81 W |
| DXTC | 1.67 W | 1.15 W |
| BFS | 0.98 W | 1.07 W |
| Histogram | 0.91 W | 1.19 W |

(b) Idle and static power of each GPU

| GPU$_k$ | $P^s_k$ | $P^{idle}_k$ |
|---|---|---|
| $\pi_0$ (RTX 3070) | 46 W | 0.445 W |
| $\pi_1$ (T400) | 8 W | 0.652 W |

consumption from the 3.3V line of PCIe was not considered because it was negligibly small (the current was less than 30 mA) and it was not substantially affected by the current state of the GPU. More details can be found in our tool demonstration paper [35].

### B. Benchmarks and Power Profiles

Six benchmark programs are considered in our experiments: MATRIXMUL, STEREODISPARITY, DXTC, HISTOGRAM from NVIDIA CUDA 11.6 Samples [36], and HOTSPOT, BFS from the Rodinia GPU benchmark suite [37]. This choice is made based on whether the execution time of the program is long enough on both GPUs for the sampling rate of our power monitoring tool or whether the input size is configurable to increase the execution time. Each program is then modified to use spatial multitasking on a separate CUDA stream, but within the same CUDA context to enable concurrent execution of these streams. The software environment we used is Ubuntu 18.04 and CUDA 11.6 SDK.

To explore the impact of different scheduling policies on these workloads, we measured their execution time and power parameters using our setup shown in the previous subsection. Fig. 2 depicts the WCET of each benchmark as the number of SMs changes on the two GPUs considered. Note that we took the maximum observed execution time as the WCET. On RTX 3070, Although the execution time of some programs appears to plateau on RTX 3070 after a certain number of SMs, it in fact decreases in proportion to the SM count. When the same number of SMs is used, RTX 3070 gives shorter execution time as it uses a newer architecture running at a higher frequency, but the ratio of the difference varies by benchmarks.

Table II shows the dynamic power parameters of the benchmarks and the idle and static power of the two GPUs. $\pi_0$ is RTX 3070 and $\pi_1$ is T400. For dynamic power, we report only the case of SM count $m_i = 1$, i.e., $P^d_{k,i}(1)$, because $P^d_{k,i}(m_i) = m_i \cdot P^d_{k,i}(1)$ holds as discussed in Sec. III-B. Interestingly, RTX 3070 does not always consume more dynamic power than T400 despite its higher frequency. Idle power is lower in RTX 3070, probably due to its newer architecture. Static power is significantly higher on RTX 3070 but this does not affect the energy consumption of the entire system unless the GPU device is unplugged or put in a deep sleep mode.

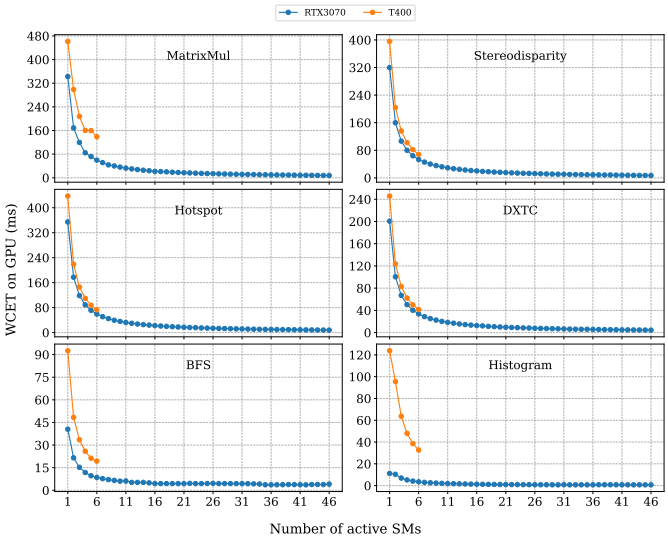



Figure 2: WCET of benchmarks on RTX 3070 and T400

### C. Observations

Based on real execution time and power parameters, we now give some examples to make observations and gain insights towards energy-efficient scheduling on a multi-GPU system.

**Baseline scheduling approaches.** Let us consider two workload allocation approaches that are well understood in the context of multiprocessor systems.

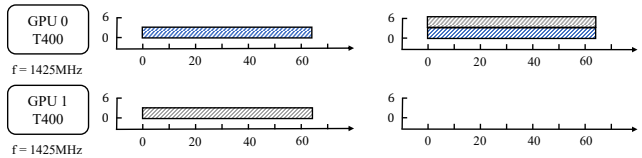
- Load Concentration: Assigns given workloads to the same resource until it gets fully utilized. For GPUs with spatial multitasking, this means a GPU task is assigned to the most packed GPU, with the remaining SMs of that GPU. This is the default allocation approach of the NVIDIA driver when the system has multiple GPUs.
- Load Distribution: Uniformly distributes given workloads across available resources. Hence, it chooses an idling GPU first (or a GPU with the highest number of unused SMs when spatial multitasking is considered). Note that this is the expected behavior when $k$-exclusion locking protocols are used [10].

In the following examples, we show how the choice of workload allocation contributes to the energy consumption of the resulting schedule. The task parameters used in the examples are extracted from the results shown in Fig. 2 and summarized in Tables III and IV. For ease of presentation, we focus on kernel execution time, $G^e_i$, and omit data copy time.

**Homogeneous GPUs.** Consider a homogeneous multi-GPU system $\Pi = \{\pi_0, \pi_1\}$ containing two identical NVIDIA T400 GPUs, i.e., $type(\pi_0) = type(\pi_1)$.

**Example 1.** Consider two tasks with the execution time parameters given in Table. III. The tasks are running on different CUDA streams, so asynchronized memory copy and current kernel execution can happen. For each GPU, a single execution instance is created for each task so that different GPUs can be used simultaneously.

To emulate a scenario that the system is lightly loaded, the number of SMs each task can use is limited to 3 on T400.




(a) Schedule w/ distributed load: E=2.3J (b) Schedule w/ concentrated load: E=2.05J

Figure 3: Scheduling results in Example 1

We select an observation window of 100ms for the following two possible schedules shown in Fig. 3: schedules by load distribution and by load concentration. In Fig.3a, the job of $\tau_1$, $J_{1,1}$, and the job of $\tau_2$, $J_{2,1}$, are distributed to two GPUs, and the estimated energy consumption of this schedule is 2.3J computed by Eq. (4). Fig.3b shows the schedule under load concentration strategy. In this schedule, $J_{1,1}$ and $J_{2,1}$ share the GPU $\pi_0$ while leaving $\pi_1$ idle so that it can be power gated. The estimated energy consumption of the system is 2.05J.

Table III: Taskset in Examples 1 and 2

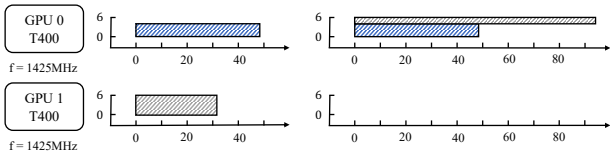
| Task | Application | $G_i^e(\pi_0, 6)$ | $G_i^e(\pi_0, 4)$ | $G_i^e(\pi_0, 3)$ | $G_i^e(\pi_0, 2)$ |
|---|---|---|---|---|---|
| $\tau_1 = \tau_2$ | Histogram | 32.67 ms | 47.95 ms | 63.724 ms | 95.53 ms |

Although the above example shows that load concentration (i.e., packing tasks to as few GPUs as possible while keeping the other GPUs idle so that they can be power gated) may be more energy efficient, it is not always true. As mentioned in [6], the use of spatial multitasking can lead to energy inefficiency since the GPU is not SM-level power-gated and unused SMs incur idle power consumption when the GPU remains active. In the next example, we will show that packing tasks to one GPU while leaving the other idle can be less energy efficient than distributing tasks to all GPUs, especially when it is inevitable to leave idle SMs for a long time.

**Example 2.** Consider the same tasks as in Examples 1. Now, the job of $\tau_1$, $J_{1,1}$, executes on $\pi_0$ with 4 SMs, and the execution time parameters are given in Table III. In the schedule shown in Fig. 4a, $J_{1,1}$ and $J_{2,1}$ are distributed to $\pi_0$ and $\pi_1$, and $J_{2,1}$ executes on $\pi_1$ with 6 SMs. In the schedule in Fig. 4b, $J_{2,1}$ is assigned to $\pi_0$ with the remaining SMs and executes in with $J_{1,1}$ concurrently, while $\pi_1$ stays idle. However, after $J_{1,1}$ finishes execution, $J_{2,1}$ is still running, during which the idle SMs on $\pi_0$ keep consuming energy, and this makes it less energy efficient than the schedule with load distribution. With Eq. (4), we can calculate the estimated energy consumption of two schedules in an observation window of 100ms, and they are 2.12J and 2.18J respectively.

**Heterogeneous GPUs.** In the next two examples, we will explore the energy consumption under two allocation approaches in a heterogeneous multi-GPU system $\Pi = \{\pi_0, \pi_1\}$ (i.e.m $type(\pi_0) \neq type(\pi_1)$). This is the same hardware configuration as in Sec. IV-A.

**Example 3.** Consider a taskset with parameters listed in Table IV. Suppose at $t = 0$, the job of $\tau_i$, $J_{1,1}$, has just




(a) Schedule w/ distributed load: E=2.12J (b) Schedule w/ concentrated load: E=2.18J

Figure 4: Scheduling results in Example 2

Table IV: Taskset in Example 3 and 4

| Task | Application | $G_i^e(30, \pi_0)$ | $G_i^e(16, \pi_0)$ | $G_i^e(6, \pi_1)$ |
|---|---|---|---|---|
| $\tau_1$ | MatrixMul | 11.98 ms | 21.55 ms | - |
| $\tau_2$ | Hotspot | 12.00 ms | 22.31 ms | 73.188 ms |

started its kernel execution on the GPU $\pi_0$ with 16 SMs, and at the same time, the job of $\tau_2$, $J_{2,1}$, is ready for execution. Following the work distribution approach, $J_{2,1}$ will execute on the GPU $\pi_1$ and the resulting schedule of the two tasks is shown in Fig. 5a. Similar to the previous examples, when an observation window of 100ms is considered, the energy consumption of this schedule is calculated to be 7.35J.

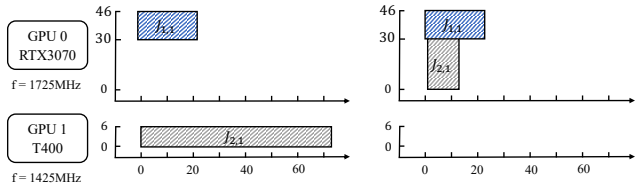
Fig. 5b shows the schedule under the load concentration approach. Since $J_{1,1}$ is not using all the SMs of the GPU $\pi_0$, $J_{2,1}$ is able to use the remainder. In this way, $\pi_1$ is idle so that it can perform power gating to save energy and the estimated energy consumption of this schedule is 7.24J.

**Example 4.** Consider the same multi-GPU system and task parameters as in Example 3. But at this time, $J_{1,1}$ starts kernel execution with 30 SMs on $\pi_k$. Following the load concentration approach, $J_{2,1}$ uses the remaining 16 SMs on $\pi_k$ as shown in Fig. 6b and the estimated energy consumption of this schedule is 7.3J. Since $\pi_1$ is idle when $J_{2,1}$ is ready for its execution, the load distribution approach executes $J_{2,1}$ on $\pi_2$ with all the available SMs. Fig. 6a shows this schedule and the estimated energy consumption here is 7.19J, which is smaller than that with the load concentration approach.

To summarize, the above examples suggest that neither load concentration nor distribution should be preferred over the other when making scheduling decisions in a multi-GPU system, regardless of whether GPUs are homogeneous or not. One thing we can clearly observe is that, if all tasks assigned to the same GPU have similar finish time, this could be helpful to reduce active-idle power consumption of unused SMs. However, this is hard to realize with real-time tasks since they have different periods and arrival patterns and their absolute completion time is determined only at runtime. The difficulty of this problem multiplies when timing constraints are considered.

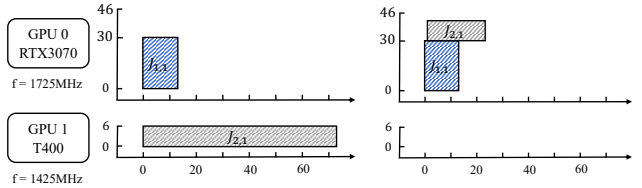
## V. Energy-Efficient Multi-GPU Scheduling

Based on the observations from the previous section, we propose our scheduling framework that makes runtime scheduling decisions for both timeliness and energy efficiency in multi-GPU systems. This approach extends sBEET [6], which is the latest work on energy-efficient real-time GPU




(a) Schedule w/ distributed load: E=7.35J (b) Schedule w/ concentrated load: E=7.24J

Figure 5: Scheduling results in Example 3




(a) Schedule w/ distributed load: E=7.19J (b) Schedule w/ concentrated load: E=7.3J

Figure 6: Scheduling results in Example 4

scheduling for a single GPU system. Hence, we name our framework as "sBEET-mg".

### A. Energy Optimality

To better explain the proposed scheduling framework, here we revisit the definition of the *energy-optimal* number of SMs given in [6] and give the definition of *energy-preferred* GPU for each task in a multi-GPU system.

**Definition 1** (Energy optimal SMs [6])**.** The energy-optimal number of SMs $m_{k,i}^{opt}$ for a task $\tau_i$ on a GPU $\pi_k$ is defined as the number of SMs that leads to the lowest energy consumption computed by Eq. (3) when it executes in isolation on the GPU $\pi_k$ during an arbitrary time interval $\delta \geq \max_{m \leq M_k} G_{i,j}^e(m, \pi_k)$.

In the above definition, it is worth noting that the energy-optimal number of SMs is unaffected by the duration of $\delta$. This is derived from Eq. (3). Assume the minimum possible $\delta_{min} = \max_{m \leq Mk} G_i^e, j(m, \pi_k)$, which is long enough for $\tau_i$ to complete execution no matter how many SMs are allocated. After $\delta_{min}$, the GPU is power-gated and only $P^s$ contributes to energy consumption under any SM allocation. Using this and the energy consumption model in Eq. (3), we can define the energy-preferred GPU as below.

**Definition 2.** (Energy preferred GPU) The energy-preferred GPU for a task $\tau_i$ in a multi-GPU system $\Pi$ is given by:

$$\underset{\pi_k \in \Pi}{\operatorname{argmin}} \int_0^{\delta} P_k^s + P_{k,i}^d(m_{k,i}^{opt}) + P_k^{idle}(M_k - m_{k,i}^{opt})dt \quad (5)$$

where $\delta$ is an arbitrary time interval ($\delta \geq \max G_{i,j}^e(\pi_k, m)$) and $m_{k,i}^{opt}$ is the energy-optimal number of SMs for $\tau_i$ on the GPU $\pi_k$. This gives the GPU that consumes the least amount of energy when $\tau_i$ executes with $m_{k,i}^{opt}$ SMs on it.

### B. Overview of sBEET-mg

The main idea of sBEET-mg is to adaptively select the GPU and the SM configuration for individual jobs of real-time tasks. When a job is arrived or completed, among all possible assignments, the scheduler chooses the one that the job can bring the minimum expected energy consumption to all GPUs in the system.

The software framework structure of sBEET-mg is similar to that of sBEET, except that sBEET-mg is specifically designed to handle multiple GPUs. The sBEET-mg framework maintains one centralized server in the system and multiple worker threads for each GPU. The role of the central server is to receive jobs from GPU tasks and let them share the same CUDA context for concurrent stream execution. Once the scheduling algorithm of the server determines the target GPU to dispatch a job, it sends the job to the corresponding worker thread for execution on that GPU. Then this worker thread calls `cudaSetDevice()` to set the GPU device to use and launches the kernel in a separate CUDA stream. With this design, a separate execution instance is available for each running job so that multiple GPUs can be utilized simultaneously. We adopt *persistent threads* for GPU partitioning. In this way, sBEET-mg enables parallel kernel execution on all GPUs in the system, and the decision on which GPU to use and when to use spatial multitasking is made by our scheduling algorithm presented later.

When the sBEET-mg framework starts, the procedure in Alg. 1 is executed to allocate tasks to GPUs. More details on this procedure will be explained below. Following the observation in [6], we limit the number of worker threads on each GPU to two since more parallelism does not necessarily improve performance [6]. Hence, the server creates two worker threads as well as two CUDA streams for each GPU, and each worker is bounded to one CUDA stream. When the worker thread receives a job, it runs that job on the corresponding CUDA stream. Each worker shares the status of its SM, i.e., active or idle, with the server through a global shared data structure whenever a job assigned to it begins and completes execution. This allows the server to have a global view of the system and make scheduling decisions properly.

Whenever a new job $J_{i,j}$ arrives, the server invokes the runtime scheduling algorithm given in Alg. 2 (explained later) to decide whether to execute this job on the preassigned GPU by Alg. 1 or migrate it to another GPU. When a job completes, the server is notified by the corresponding worker and freed SM resources are reclaimed for the execution of next or pending jobs.

### C. Offline Task Distribution

For a given taskset $\Gamma$, the proposed task distribution algorithm allocates tasks to GPUs offline. Basically, for each task $\tau_i \in \Gamma$, the algorithm tries to assign it to the *energy-preferred* GPU $\pi_x$ with $m_{x,i}^{opt}$ as long as the capacity of $\pi_x$ permits. Alg. 1 depicts the pseudocode of the task distribution procedure. It first sorts all tasks in $\Gamma$ in decreasing order of priority so that higher-priority tasks have a better chance to get

**Algorithm 1** Offline Task Distribution

```
1: procedure TASK DISTRIBUTION
2:     Sort tasks in Γ in decreasing order of priority
3:     for τ_i ∈ Γ do
4:         Get a list Π_i of GPUs in non-increasing order of expected energy consumption for τ_i
5:         for π_k ∈ Π_i do
6:             if U(π_k) + U_i(π_k, m^opt_{k,i}) ≤ 1 then
7:                 Assign τ_i to π_k
8:                 break
9:             end if
10:        end for
11:        if τ_i is not assigned then
12:            Assign τ_i to the GPU that has a minimum utilization after τ_i is assigned
13:        end if
14:    end for
15: end procedure
```

their energy-preferred GPUs (line 2). Then for each task $\tau_i$, it obtains a list $\Pi_i$ of GPUs in non-increasing order of expected energy consumption. Hence, the energy-optimal GPU of $\tau_i$ goes first in this list. For each GPU $\pi_k$ in the ordered list $\Pi_i$, it runs a simple utilization check to decide whether $\tau_i$ can be accepted (line 3 to line 10). After iterating through all the GPUs, if $\tau_i$ is still not assigned to any GPU, the algorithm assigns it to the GPU that will have the minimum utilization after $\tau_i$ is assigned (line 12). The result of this allocation serves as a guideline for the runtime scheduler.

### *D. Runtime Job Migration*

Alg. 1 gives an offline task distribution strategy, and this can lead to an energy-efficient schedule if all tasks can execute on its energy-preferred GPU with the optimal number of SMs. However, according to the given examples and the previous work [7], it might not be energy efficient to turn on multiple GPUs when the system is underutilized, since the GPUs are not SM-level power gated and the energy consumed by active-idle SMs can negatively affect the total energy consumption of the system. Therefore, we seek opportunities to further reduce the energy consumption of a multi-GPU system by judiciously migrating and packing jobs at runtime.

Before introducing the proposed algorithm, we write a function to adopt and encapsulate some methods of sBEET (Alg. 2 and 3 in [6]):

**function:** sBEET($\pi_k$, $J_{i,j}$); **returns** ($S^{cfg}_{i,j}(\pi_k)$, $E$)

The function sBEET takes two inputs, $\pi_k$ and $J_{i,j}$, where

- $\pi_k$ is the GPU that the caller (the runtime scheduler of sBEET-mg, namely Alg. 1) wants to check.
- $J_{i,j}$ is the job that the caller is going to make a scheduling decision for.

It returns a tuple of $S^{cfg}_{i,j}$ and $E$. $S^{cfg}_{i,j}$ is the SM allocation result on $\pi_k$ for $J_{i,j}$. If $S^{cfg}_{i,j} = \emptyset$, $J_{i,j}$ cannot execute on $\pi_k$ for now. $E$ is the expected energy consumption of $\Pi$ during a time window from the current time to the estimated finish time of $J_{i,j}$, $f_{i,j}$, with the SM allocation $S^{cfg}_{i,j}$. Hence, by Eq. (4), $E$ is equal to $E([t_{now}, f_{i,j}])$. If $S^{cfg}_{i,j} = \emptyset$, $E = \infty$.

Alg. 2 gives the proposed runtime job migration scheduler. It takes as input a job $J_{i,j}$ which is either a newly-released job (if there is no other pending job) or the highest-priority pending job. The algorithm decides whether the job should be launched at the current time or be delayed, which GPU to use, and how many SMs should be assigned, by considering the current status of the task's energy-preferred GPU $\pi_x$. As a result of scheduling decision making, the algorithm returns a SM configuration $S^{cfg}_{i,j}(\pi_k)$ for $J_{i,j}$. If $S^{cfg}_{i,j}(\pi_k) = \emptyset$, $J_{i,j}$ is pushed to the pending queue for later consideration.

- (Alg. 2 line 2 to 24) If $\pi_x$ is idle, the scheduler tentatively puts $J_{i,j}$ on $\pi_x$ with $m^{opt}_{x,i}$ SMs, checks if $J_{i,j} \cup \pi_x$ can meet their deadlines,[4] and then estimates the energy consumption that $J_{i,j}$ will contribute to the whole system. If $J_{i,j} \cup \pi_x$ are not expected to meet deadlines, the computed energy $E_1$ is set to $\infty$, meaning the assignment is invalid (line 2 to 9). Then the scheduler will check whether there is any chance to follow the packing strategy to launch $J_{i,j}$ on other active GPUs so that $\pi_x$ can be power gated to save energy. It iterates through $\Pi_i$ which is obtained in Alg. 1, and follows the method in sBEET to find whether there is any assignment that can be more energy efficient and reduce deadline violations by exploiting spatial multitasking techniques. The predicted energy will be saved as $E_2$, and the scheduler will return the $S^{cfg}_{i,j}$ with the smaller predicted energy consumption (line 10 to 24).
- (Alg. 2 line 25 to 27) In the second case, $\pi_x$ is partially occupied. The scheduler calls sBEET($\pi_k$, $J_{i,j}$) to decide and return the SM configuration $S^{cfg}_{i,j}$.
- (Alg. 2 line 28 to 46) If $\pi_x$ is fully occupied, we consider the following two cases: (i) $J_{i,j}$ can be postponed and wait for $m^{opt}_{x,i}$ SMs on GPU $\pi_x$ (line 29 to 36), or (ii) execute on a GPU other than $\pi_x$ (line 37 to 43). In case (i), the scheduler estimates the time when $\pi_x$ would become available with $m^{opt}_{x,i}$ SMs. If $J_{i,j}$ can meet the deadline with this assignment, then the scheduler predicts the energy consumption from the current time to the estimated finish time of $J_{i,j}$. Otherwise, the computed energy $E_4$ is set to $\infty$. In case (ii), the scheduler iterates through $\Pi_i$ and predicts the energy consumption if $J_{i,j}$ is placed on a GPU other than $\pi_x$. For each $\pi_{k \neq x} \in \Pi_i$, if $\pi_k$ is not fully occupied, the scheduler calls sBEET($\pi_k$, $J_{i,j}$) to get SM configuration $S'_5$ and the predicted energy consumption $E'_5$. After all the available GPUs are traversed, the scheduler saves the configuration with minimum energy consumption. After these procedures are done, the scheduler returns the corresponding SM allocation $S^{cfg}_{i,j}$ of (i) or (ii) that leads to smaller energy consumption of $J_{i,j}$'s execution.

[4] This is done by following the original sBEET's approach (Alg. 3 in [6]) that generates a schedule from the current time to $f_{i,j}$ for a given SM allocation on $\pi_x$ and checks if all jobs can meet their deadlines until $f_{i,j}$.

**Algorithm 2** Runtime Job Migration

1: **function** JOB MIGRATION($J_{i,j}$)
2: **if** $\pi_x$ is idle **then**
3: Tentatively place $J_{i,j}$ on $\pi_x$ with $m_{x,i}^{opt}$ SMs
4: **if** $J_{i,j} \cup \pi_x$ will meet deadlines **then**
5: $E_1 \leftarrow E([t_{now}, f_{i,j}])$
6: $S_1 \leftarrow$ the corresponding SM allocation
7: **else**
8: $E_1 \leftarrow \infty$
9: **end if**
10: $E_2 \leftarrow \infty$
11: **for** each $\pi_{k \neq x}$ in $\Pi_i$ sorted by Alg. 1 **do**
12: **if** $\pi_k$ is idle or $\pi_k$ is fully occupied **then**
13: **continue**
14: **else**
15: $(S_2', E_2') \leftarrow$ sBEET($\pi_k$, $J_{i,j}$)
16: $E_2 \leftarrow \min(E_2, E_2')$
17: **end if**
18: **end for**
19: **if** $E_1 == \infty$ and $E_2 == \infty$ **then**
20: Assign $J_{i,j}$ with maximum SMs on $\pi_x$
21: **else**
22: Select the schedule with $\min(E_1, E_2)$
23: **end if**
24: **return** $S_{i,j}^{cfg}$ ▷ the corresponding SM allocation for $J_{i,j}$
25: **else if** $\pi_x$ is partially occupied **then**
26: $(S_{i,j}^{cfg}, E) \leftarrow$ sBEET($\pi_x$, $J_{i,j}$)
27: **return** $S_{i,j}^{cfg}$
28: **else** ▷ If the GPU is full
29: $t_1 \leftarrow$ current time
30: Tentatively place $J_{i,j}$ on $\pi_x$ and wait until $m_{x,i}^{opt}$ SMs become available
31: $t_2 \leftarrow f_{i,j}$
32: **if** $J_{i,j} \cup \pi_x$ will meet deadlines **then**
33: $E_4 \leftarrow E([t_1, t_2])$
34: **else**
35: $E_4 \leftarrow \infty$
36: **end if**
37: $E_5 \leftarrow \infty$
38: **for** each $\pi_{k \neq x}$ in $\Pi_i$ sorted by Alg. 1 **do**
39: **if** $\pi_k$ is not full **then**
40: $(S_5', E_5') \leftarrow$ sBEET($\pi_x$, $J_{i,j}$)
41: $E_5 \leftarrow \min(E_5, E_5')$
42: **end if**
43: **end for**
44: Select the schedule with $\min(E_4, E_5)$
45: **return** $S_{i,j}^{cfg}$ ▷ the corresponding SM allocation for $J_{i,j}$
46: **end if**
47: **end function**

### E. Time Complexity

According to the time complexity analysis in [6], the time complexity of the original sBEET is $O(n \cdot log(n))$ where $n$ is the number of tasks. Suppose the number of GPUs in the system is $\omega$. In Alg. 2, the procedure to check whether a job can be scheduled on each GPU (lines 11 to 18 and 38 to 43) is upper-bounded by $\omega \cdot O(n \cdot log(n))$; hence, the time complexity of the runtime job migration is given by $O(\omega \cdot n \cdot log(n))$.

### F. Offline Schedule Generation

This work targets soft real-time systems with no hard guarantees. Tasks are always accepted, and our algorithms try to minimize deadline misses and energy consumption. If one needs hard guarantees, our algorithms can be used to generate a schedule for one hyperperiod offline, check if this meets all deadlines, and run it as a time-triggered schedule at runtime.

## VI. EVALUATION

This section carries out experiments using our implementation for real hardware setup as well as simulation.[5]. The majority of experiments are conducted on the hardware setup given in Sec. IV-A. To evaluate performance in systems with more GPUs, we also present experimental results from a Python simulator we developed (Sec. VI-B).

In both experimental setups, we compare the performance of sBEET-mg against the following approaches: (i) "LCF" (Little-Core-First) with LTF (Largest-Task-First), (ii) "BCF" (Biggest-Core-First) with LTF, both of which represent the load concentration approach, and (iii) "Load-Dist" (load distribution).[6] We also consider "sBEET-mg Offline Only" to assess the effect of the runtime algorithm (Alg. 2).

### A. Hardware Experiments

In all the experiments on real hardware, we use the system shown in Fig. 1 consisting of two GPUs, RTX3070 and T400. Since the difference in computational power between these two GPUs is too large, we decided to use only a portion of SMs on RTX3070. This is reasonable since in practice, there is a possible scenario where a portion of the GPU can be reserved for the dedicated use of high-critical tasks, and the remaining is shared among other tasks. In this system, $\Pi = \{\pi_0, \pi_1\}$, where $type(\pi_0) =$ RTX3070 and $type(\pi_1) =$ T400. We set $\pi_0$ as the reference GPU, and the utilization of each task ($U_i(\pi_k) = U_i(\pi_0)$) can be determined in this way.

Table V: Parameters for taskset generation

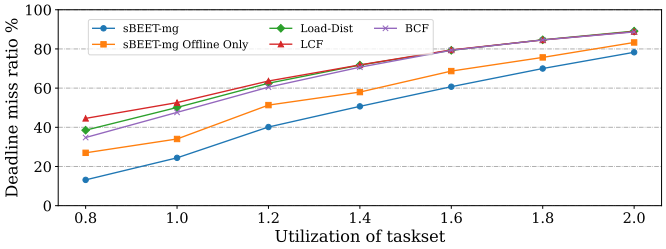
| Parameters | Range |
|---|---|
| Workload of the task | One of the eight mentioned benchmarks |
| Number of tasks | 6 |
| $U_i(\pi_0)$ | [0.01, 0.5] |
| $D_i$ | 0.5 * $T_i$ |

*1) Results of Schedulability:* In this experiment, we compare the schedulability of the proposed method with the other approaches. For each value of utilization, 100 tasksets are randomly generated with the parameters given in Table V using the UUnifast algorithm [38], and we use RM to decide task priorities. For each taskset, we run each approach for 15 seconds, and measure the deadline miss ratio of the tasks. Due to the reason mentioned in Sec. VI-A, we limit the number of SMs to be used on RTX3070 to 6, 12 and 24, and run the same tasksets on the respective SM configurations.

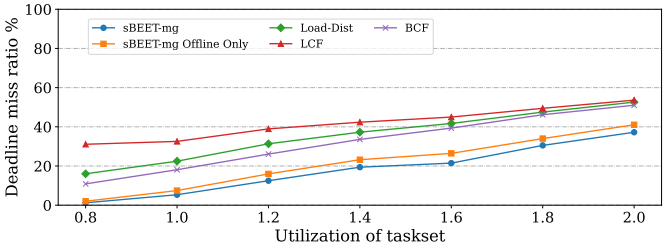
Fig. 7 presents the absolute runtime deadline miss ratio under each method, and sBEET-mg always has the lowest deadline miss ratio among them. In particular, sBEET-mg achieves up to 23% and 18% reduction in deadline misses compared to Load-Dist and BCF, respectively. Since we use the same tasksets in all the cases, the system gets most heavily

[5]Source code is available at https://github.com/rtenlab/sBEET-mg/.

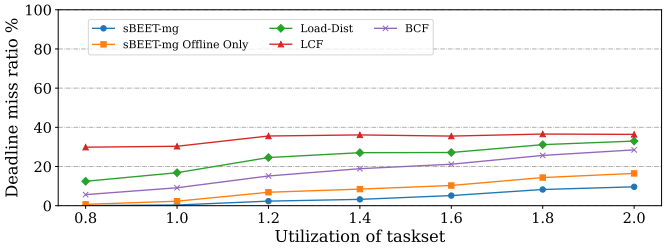
[6]We define a "big-core" as a GPU with higher computational capacity and a "little-core" as a GPU with lower capacity, i.e. more and fewer SMs.




(a) 6 SMs allowed on RTX3070

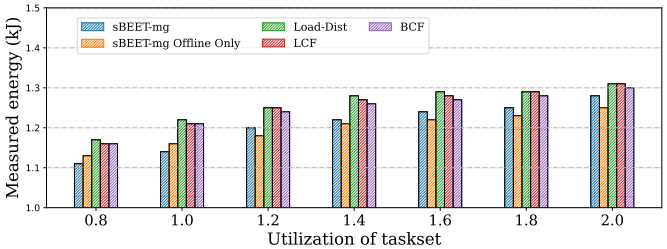



(b) 12 SMs allowed on RTX3070

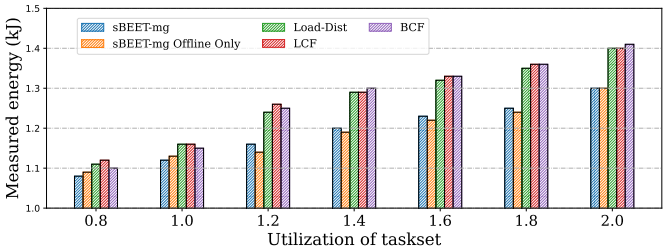



(c) 24 SMs allowed on RTX3070

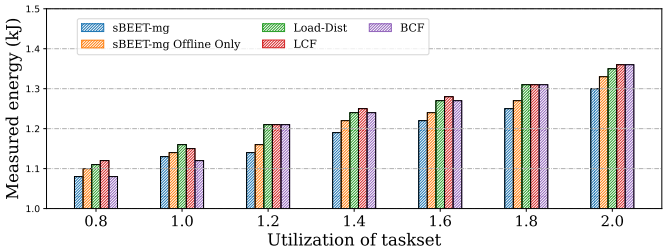
Figure 7: Deadline miss ratio w.r.t. the utilization of taskset




(a) 6 SMs allowed on RTX3070




(b) 12 SMs allowed on RTX3070




(c) 24 SMs allowed on RTX3070

Figure 8: Energy consumption w.r.t. taskset utilization

loaded when SMs on RTX3070 is limited to 6 as Fig. 7a shows, and least loaded when SMs on RTX3070 is limited to 24 as Fig. 7c shows. We can see that the deadline miss ratio under all methods is getting lower from the top figure to the bottom. In Fig. 7a, all the curves are closer to each other since both of the GPUs only have 6 SMs that are allowed to be used. Due to this reason, there is not much space for sBEET-mg to play around. However, as more SMs are allowed on RTX3070 as shown in Fig. 7b and 7c, especially as the system gets overloaded, since our proposed method takes into account the future arrival of the tasks to find the right GPU and the number of SMs, the tasks will have less chance get starved, our method can significantly reduce deadline miss ratio.

*2) Results of Energy Consumption:* While running the experiments in Sec. VI-A1, we also measured the runtime energy consumption of the five approaches, and the results are shown in Fig. 8. At first, we can observe that, with 24 SMs on RTX3070 and $U \leq 1.0$, BCF yields marginally better energy consumption than sBEET-mg. This is because BCF assigns all workloads to the bigger GPU (RTX3070) and leaves the smaller GPU (T400) idle all the time; however, it causes an excessively high number of deadline misses, as shown in Fig. 7c. In the other cases, the energy consumption of sBEET-mg and sBEET-mg Offline Only is always lower than the other three approaches that are energy-agnostic. Under all the three SM configurations with $U \leq 1.0$, the energy consumption of sBEET-mg is lower than sBEET-mg Offline Only. The reason is, when the system is not overloaded, the job migration algorithm has more chances to take effect to save energy. Also, sBEET-mg is always more energy-efficient than sBEET-mg Offline Only when 24 SMs are used on RTX3070. With 6 and 12 SMs enabled on RTX3070 and $U \geq 1.2$, the energy consumption of sBEET-mg Offline Only is the lowest because (1) it guarantees that the tasks always run with $m^{opt}$, and (2) in sBEET-mg, the use of the runtime algorithm with spatial multitasking and job migration improves schedulability, which inevitably causes more energy consumption [6].

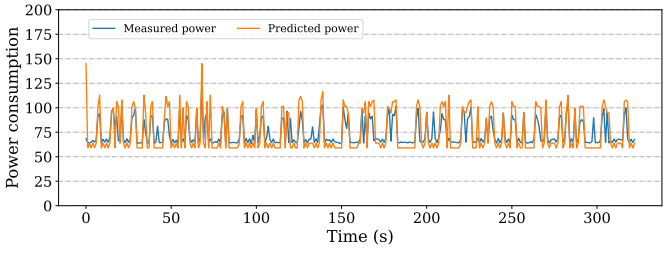



Figure 9: Trace of actual and predicted power consumption

*3) Power Prediction Accuracy:* To evaluate the effectiveness of the power prediction method used in our proposed scheduler, we compare the predicted power consumption with the actual power consumption measured by the power monitoring tool in Sec. IV-A. For each utilization considered, we randomly select one taskset consisting of 6 tasks from the

Table VI: Power prediction for tasksets with different utilizations

** Note: Maximum power is ≈ 180 W*

| Taskset Util. | $E_{meas}$ (kJ) | $E_{pred}$ (kJ) | $MAE_{power}$ (W) | Released job | Missed job |
|---|---|---|---|---|---|
| 0.8 | 21.53 | 21.70 | 10.79 | 8882 | 1 |
| 1.0 | 22.12 | 22.71 | 9.56 | 13174 | 1 |
| 1.2 | 21.91 | 23.14 | 8.33 | 11858 | 0 |
| 1.4 | 22.30 | 24.05 | 10.55 | 16909 | 4 |
| 1.6 | 23.14 | 22.92 | 10.53 | 18033 | 438 |
| 1.8 | 24.16 | 24.84 | 11.54 | 23173 | 456 |
| 2.0 | 26.36 | 27.84 | 14.27 | 25841 | 865 |

benchmark pool using the parameters given in Table V, and run each taskset using our proposed scheduler for 5 minutes. Fig. 9 illustrates the measured and estimated power traces of a taskset with utilization of 1.0. Table. VI summarizes the results from all tasksets tested: $E_{meas}$ and $E_{pred}$ stand for measured and predicted energy, respectively, and $MAE$ is the mean-absolute-error (MAE) in power prediction. The numbers of jobs released and missed deadlines during measurement are also reported. The average MAE of all the tasksets of different utilization is 10.80 W ($\approx 6\%$ of 180 W), and we can say the power prediction accuracy is good enough for this work.

*4) Comparison with sBEET:* One may wonder how the original sBEET would perform if it is used in a multi-GPU system with conventional offline task allocation methods such as BFD, WFD, and FFD. In this experiment, we answer this question by comparing the schedulability of the proposed work against the original sBEET combined with three allocation methods. The tasksets generated with the parameters in Table V are used, and the number of SMs is set to 24 on RTX3070. For each taskset, we run sBEET-mg, sBEET Offline Only, WFD + sBEET, FFD + sBEET and BFD + sBEET for 15 seconds each, and measure the deadline miss ratio. Fig. 10 presents the absolute deadline miss ratio under the five approaches, and sBEET-mg has the lowest among all of them. Note that the curves of FFD + sBEET and BFD + sBEET are overlapped because they had the exact same performance in our experiments.

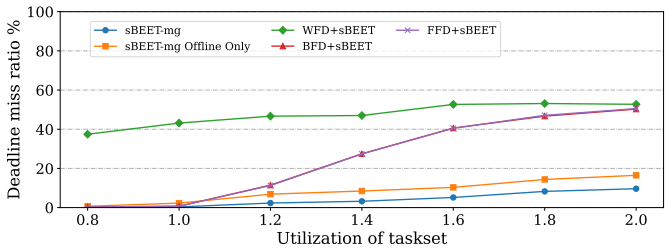


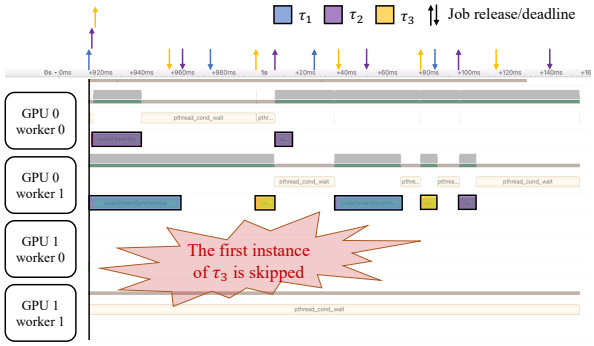

Figure 10: Deadline miss ratio of sBEET-mg and sBEET

*5) Effect of Job Migration:* To better understand the effect of runtime job migration, let us consider the following two case studies.

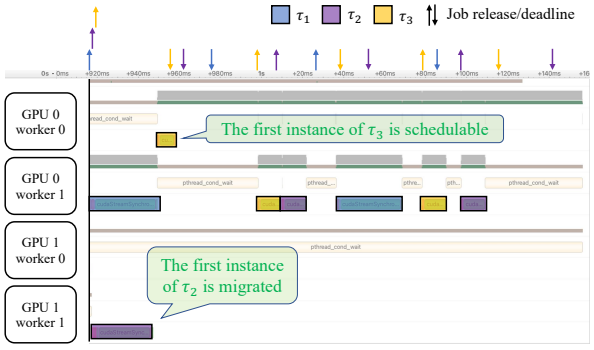
**Case Study 1.** Fig. 11 depicts the execution traces of the taskset listed in Table VII under sBEET-mg with and without job migration. The trace was collected using NVIDIA Nsight Compute. The task-related GPU activities are highlighted in different colors. For this taskset, all tasks are assigned to RTX3070 by Alg. 1 due to the energy efficiency consideration. However, they are not schedulable when job migration is not used; as noted in Fig. 11a, $J_{3,1}$ is skipped. Fig. 11b shows the case where job migration is enabled. Unlike the previous case, when $J_{1,1}$ arrives, the line 4 of Alg. 2 finds that the schedule would not be feasible if $J_{1,1}$ is executed with $m^{opt}$. Hence, it jumps to line 20 and runs $J_{1,1}$ as fast as possible on RTX3070. Later when $J_{2,1}$ arrives, as RTX3070 is fully occupied by $J_{1,1}$, line 32 takes effect and finds $J_{2,1}$ would miss the deadline if it waits until RTX3070 becomes idle. The algorithm further looks for opportunities to run $J_{2,1}$ on other GPUs and decides to move $J_{2,1}$ to T400. In this way, all three jobs are schedulable.

Table VII: Taskset used in case study 1

| Task | $D_i = 0.5 * T_i$ (ms) | Offset (ms) | GPU assigned by Alg. 1 |
|---|---|---|---|
| $\tau_1$ | 60 | 0 | RTX3070 |
| $\tau_2$ | 45 | 1 | RTX3070 |
| $\tau_3$ | 40 | 2 | RTX3070 |




(a) sBEET-mg w/o migration




(b) sBEET-mg

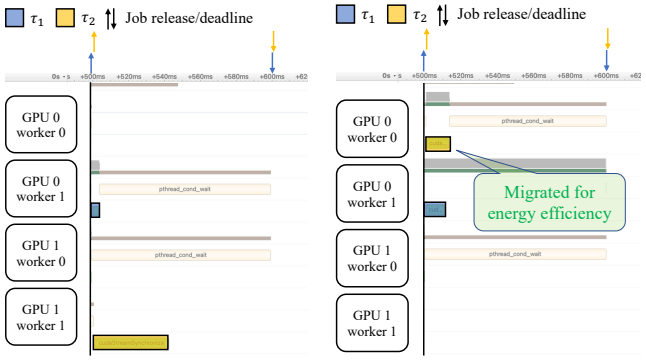
Figure 11: Job migration case study 1

**Case Study 2.** The taskset used in this case study is listed in Table VIII and the execution traces are shown in Fig. 12. For

this taskset, $\tau_1$ and $\tau_2$ are assigned to RTX3070 and T400, respectively, by Alg. 1. In Fig. 12a where migration is not used, $J_{1,1}$ and $J_{2,1}$ run on their assigned GPUs exclusively. In Fig. 12b, when $J_{2,1}$ arrives, Alg. 2 decides to move it to another GPU to run concurrently with $J_{1,1}$ for energy efficiency (line 10 to 24. We measured the energy consumption of these two schedules: the one without migration is 6.51J and the one with migration is 6.49J. Despite the small difference, this result shows the energy benefit of runtime migration.

Table VIII: Taskset used in case study 2

| Task | $D_i = 0.5 * T_i$ (ms) | Offset (ms) | GPU assigned by Alg. 1 |
|---|---|---|---|
| $\tau_1$ | 100 | 0 | RTX3070 |
| $\tau_2$ | 100 | 1 | T400 |




(a) sBEET-mg w/o migration (b) sBEET-mg

Figure 12: Job migration case study 2

### B. Simulation with Multiple GPUs

Although there are only two GPUs in our hardware setup, our proposed method can handle a system containing more GPUs, including homogeneous GPUs. We developed a simulator using Python for this purpose, in which our proposed method and the baselines are implemented.

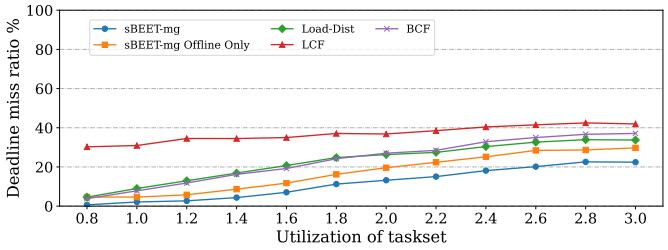
With the collected workload and power profile on the real GPUs, we add the third GPU, another RTX3070 to the simulation. In this experiment, we limit the number of SMs on both RTX3070s to 12, and the configuration is given in Table IX. With parameters given in Table V, for each taskset utilization, 200 tasksets are generated and each runs for 15 seconds. The results of the deadline miss ratio and the predicted energy consumption are demonstrated in Fig. 13. The proposed method has the best schedulability among the five methods, and in most cases, sBEET-mg and sBEET Offline Only have better energy consumption compared to the other baselines. The reason why sBEET has higher energy consumption than sBEET-mg Offline Only when $U \geq 1.6$ is due to its better schedulability, as discussed in Sec. VI-A2.

## VII. Conclusion

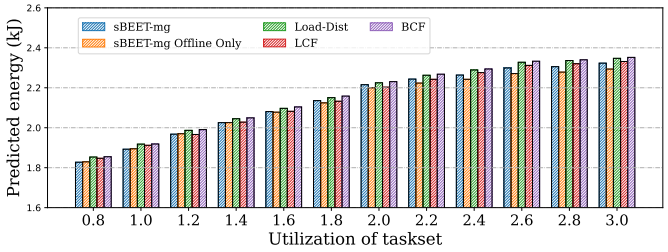
In this paper, we first provided observations about scheduling strategies in a multi-GPU system and found that existing simple task allocation approaches are not a preferred option

Table IX: GPU configurations in simulation

| GPU Id | GPU | $M_k$ | $M_k^{limit}$ |
|---|---|---|---|
| $\pi_0$ | RTX3070 | 46 | 12 |
| $\pi_1$ | RTX3070 | 46 | 12 |
| $\pi_2$ | T400 | 6 | 6 |




(a) Miss ratio w.r.t utilization of taskset




(b) Predicted energy w.r.t utilization of taskset

Figure 13: Simulation results of GPU configuration in Table IX

for energy efficiency regardless of whether GPUs are homogeneous or heterogeneous. This is mainly due to the fact that today's GPU architectures are not SM-level power-gated but device-level power-gated; thus, some unused SMs can continue to draw power although leaving as many processing units idle as possible has been considered conventional wisdom for CPU energy management. Based on these observations, we extended prior work and proposed sBEET-mg, the multi-GPU scheduling framework that improves both real-time performance and energy efficiency by assigning energy-preferred GPUs to tasks and performing job-level migration with SM-level resource allocation. The effects of sBEET-mg in reducing energy consumption and deadline miss rates are demonstrated through various experiments on real hardware and simulation.

The precise measurement and analysis of power consumption on the latest GPU architectures will give insights to future research endeavors. We hope that our findings can serve as an important stepping stone for the development of energy-efficient multi-GPU real-time systems.

## Acknowledgment

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

## References

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

[2] S. Hosseinimotlagh, A. Ghahremannezhad, and H. Kim, "On dynamic thermal conditions in mixed-criticality systems," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2020, pp. 336–349.

[3] 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)*. IEEE, 2019, pp. 254–266.

[4] Y. Lee, K. G. Shin, and H. S. Chwa, "Thermal-aware scheduling for integrated CPUs-GPU platforms," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–25, 2019.

[5] 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*, 2012, pp. 1–12.

[6] Y. Wang, M. Karimi, Y. Xiang, and H. Kim, "Balancing energy efficiency and real-time performance in GPU scheduling," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 110–122.

[7] A. Jahanshahi, H. Z. Sabzi, C. Lau, and D. Wong, "GPU-NEST: Characterizing energy efficiency of multi-GPU inference servers," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 139–142, 2020.

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

[9] 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*, vol. 88, pp. 97–109, 2018.

[10] G. Elliott *et al.*, "GPUSync: A framework for real-time GPU management," in *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[11] G. Elliott and J. Anderson, "An optimal $k$-exclusion real-time locking protocol motivated by multi-GPU systems," *Real-Time Systems*, vol. 49, no. 2, pp. 140–170, 2013.

[12] 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.

[13] 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)*, 2014, pp. 726–731.

[14] 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)*, 2014, pp. 193–204.

[15] 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*, vol. 26, no. 3, pp. 748–760, 2015.

[16] 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)*, 2019, pp. 29–41.

[17] 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)*, 2019, pp. 1–6.

[18] A. Zou, J. Li, C. D. Gill, and X. Zhang, "RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization," *arXiv preprint arXiv:2101.10463*, 2021.

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

[20] 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)*, 2019, pp. 1–5.

[21] Z.-G. Tasoulas and I. Anagnostopoulos, "Improving GPU performance with a power-aware streaming multiprocessor allocation methodology," *Electronics*, vol. 8, no. 12, 2019. [Online]. Available: https://www.mdpi.com/2079-9292/8/12/1451

[22] 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)*, 2017, pp. 104–115.

[23] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017.

[24] N. Otterness and J. H. Anderson, "AMD GPUs as an alternative to NVIDIA for supporting real-time workloads," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[25] H. Kim, P. Patel, S. Wang, and R. R. Rajkumar, "A server-based approach for predictable GPU access control," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 2017, pp. 1–10.

[26] 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)*, 2020, pp. 56–69.

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

[28] R. A. Bridges, N. Imam, and T. M. Mintz, "Understanding GPU power. a survey of profiling, modeling, and simulation methods," *ACM Computing Surveys*, vol. 49, no. 3, 9 2016.

[29] 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.*, 2003, pp. 93–104.

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

[31] "Nvidia-smi," https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf, accessed: May. 2022.

[32] M. Burtscher, I. Zecena, and Z. Zong, "Measuring GPU power with the K20 built-in sensor," in *Proceedings of Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-7. New York, NY, USA: Association for Computing Machinery, 2014, p. 28–36. [Online]. Available: https://doi.org/10.1145/2588768.2576783

[33] Texas Instrument, "Ina260 36v, 16-bit, precision i2c output current/voltage/power monitor," https://www.ti.com/product/INA260.

[34] Nordic Semiconductor, "Nrf52832 soc," https://www.nordicsemi.com/products/nrf52832.

[35] M. Karimi, Y. Wang, and H. Kim, "An open-source power monitoring framework for real-time energy-aware GPU scheduling research," in *Open Demo Session of IEEE Real-Time Systems Symposium (RTSS@Work)*, 2022.

[36] "Nvidia CUDA samples," https://github.com/NVIDIA/cuda-samples.

[37] 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)*, 2009, pp. 44–54.

[38] E. Bini, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, pp. 129–154, 05 2005.