UNIVERSITY OF CALIFORNIA
RIVERSIDE

Advancing Real-Time GPU Scheduling: Energy Efficiency and Preemption Strategies

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Yidi Wang

September 2023

Dissertation Committee:

Dr. Hyoseung Kim, Chairperson
Dr. Daniel Wong
Dr. Cong Liu

The Dissertation of Yidi Wang is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

I am grateful to my advisor, Prof. Hyoseung Kim, without whose help, I would not have been here. His academic proficiency and admirable personality have had an immeasurable impact on my PhD journey. Throughout the ups and downs of research and life, he has consistently been a source of guidance and inspiration. Whenever I felt lost or uncertain, he had a remarkable ability to instill in me a sense of purpose and determination. I am grateful for his invaluable mentorship, which extended beyond the realm of academia, for my well-being, ensuring that I maintained a healthy work-life balance. His dedication to my success and his genuine care for my well-being have been truly extraordinary. While I hold immense gratitude for his invaluable contributions, I also feel a sense of regret that circumstances prevented him from hooding me. However, I am grateful for the profound impact he has had on my life and career, regardless of this one moment. I consider myself super fortunate to have had the opportunity to work under his guidance since 2017 when I was an undergraduate, and I look forward to continuing to make him proud as I carry his guidance with me throughout my future endeavors.

I would like to express my appreciation to my labmates. Their camaraderie, collaboration and support have made my research experience enjoyable. I am grateful for the lively discussions, shared knowledge, and memorable moments we had together, especially those afternoons before the pandemic when we were having conversations about various interesting topics regardless of whether they made sense or not. Although that time was relatively short, I genuinely miss those moments. With some of them, our connection grew into something deeper, evolving into friendships that extended beyond the laboratory

walls. Thank you, my dear labmates, including Mehdi, Yecheng, Hyunjong, Mohsen, Abdul, Daniel, Louay, Hoora, Shamima, Ryan and Ziliang for making the lab not only a place of scientific growth but also a space filled with friendship and cherished experiences. I owe special thanks to Ziliang for his excellent TA work in the final quarter of my PhD journey. His dedication alleviated my burden dueing the challenging period, allowing me to focus more on teaching responsibilities and my academic work.

I am immensely grateful for the support and friendship of my dear friends throughout my journey, and their support, encouragement, and understanding have been invaluable to me. I would like to extend my heartfelt gratitude to each of them: Shuozhen, Zikui, Zeyi Jiang, Aowen Li, Yunyang Li, Yibo Liu, Xiaoxie Ma, Jie Xu, Zhibo Xu, Kexin Yi, Xinyue Zhao, Yuyang Zhan, Ziwei Zhu, and many more for spending time with me and willingness to lend a helping hand. I would also like to express my gratitude to my music teacher, George Perez, whose guidance and expertise have played a vital role in my personal and artistic development. Furthermore, I want to thank the friends I met in UCR Aikido club, although our time together may have been relatively short. Besides leaving injuries on my body, their presence has made the final year of my PhD journey remarkably memorable. Finally, I want to thank all my friends, too numerous to mention individually, for their friendship, love and encouragement throughout the journey. I am truly fortunate to have such incredible companions in my life, and I am eternally grateful for their presence.

I would like to express my most profound gratitude to my family. Throughout almost my entire PhD journey, I was unable to visit them, and for that, I am filled with a profound sense of guilt as I was not being there for them during the incredibly challenging

times of the pandemic in Wuhan, and I failed to fulfill my responsibilities over the past 3.5 years.

The memory of the day I left home in December 2019 haunts me to this day. I made a promise to my maternal grandparents, who were the closest individuals to me during the first 21 years of my life, that I would return soon in the summer. Little did I know that it would be my last embrace with my grandmother. As I complete this dissertation, I am uncertain whether I will have the opportunity to see other senior members of my family again, including my beloved sister, Duoduo Wang, the family dog. Each time I have a video chat with them, I am overwhelmed with a sense of loss and helplessness. They frequently inquire about my return home, to which I can never provide a satisfactory answer. I feel ashamed as I weakly respond, "it will be soon."

I am grateful to have family members who remain physically close. My aunt made a special trip from the east coast to Riverside to attend my PhD hooding ceremony representing the elders in my family. Additionally, I was fortunate to have my cousin present, adding to the joyous atmosphere to the big moment. At a time when my senses were overwhelmed, and I struggled to find my place amidst the ceremony, my aunt called my name from the crowd and in that moment, I felt myself was glowing.

I am forever indebted to my family for their unwavering love, support, and understanding throughout my PhD journey. Despite the physical distance that separated us, their presence in my heart and mind has been a constant source of motivation and strength. I carry the weight of those missed moments with deep remorse, deeply regretting the times when I was unable to be there for them. It is their boundless love and selfless sacrifices

that have molded me into the person I am today, and for that priceless gift, I am eternally grateful.

"May the deceased rest in peace while the living must move on with their lives." In closing, I would like to express my deepest and most heartfelt gratitude to Yezhou, the constant companion who stood by my side throughout my entire PhD journey. It is incredibly to look back at the time we have spent together. We got to know each other in 2015 at Huazhong University of Science and Technology, but it wasn't until 2018 that our acquaintance blossomed into something more profound. In 2019, he followed me to UCR. In 2022, we got a home and adopted Dwendwen Wang the dog, and in 2023, we formed our family. Yezhou has an ability to mend the shattered pieces of my being and give me the courage to confront any obstacles that come my way, for I know that "he is always there". Together, we have forged a bond that is unbreakable, and for his love, strength, and belief in me, I am eternally grateful.

To my parents for all the support.

ABSTRACT OF THE DISSERTATION

Advancing Real-Time GPU Scheduling: Energy Efficiency and Preemption Strategies

by

Yidi Wang

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, September 2023
Dr. Hyoseung Kim, Chairperson

Real-time GPU scheduling plays a critical role in meeting the performance and timing requirements of modern cyber-physical and autonomous systems, where timely execution of critical tasks is essential. However, the ever-increasing complexity and heterogeneity of modern computing hardware have introduced new challenges in achieving resource and energy efficiency while delivering the required real-time performance. With the proliferation of GPU-accelerated applications and the rise of power-constrained environments, there is an urgent need to optimize resource allocation, minimize energy consumption, and successfully execute time-sensitive tasks.

In this dissertation, we address the challenges in real-time GPU scheduling by proposing novel algorithms system-level solutions. Firstly, we investigate the trade-off between energy efficiency and real-time performance, developing novel approaches that dynamically allocate resources based on task characteristics. This optimization allows us to balance power consumption while meeting strict timing constraints. Secondly, we focus on energy-efficient real-time scheduling in heterogeneous multi-GPU systems. By consid-

ering the heterogeneity of GPU architectures and workload characteristics, we introduce strategies to improve both energy efficiency and real-time performance. Lastly, we develop systematic techniques to enable preemptive priority-based scheduling for real-time GPU tasks. Through the utilization of the proposed preemption techniques, we enhance resource utilization, improve responsiveness, and achieve enhanced schedulability in multi-core systems. Through this work, we make significant contributions to the field of real-time GPU scheduling, addressing energy efficiency, performance balancing, and preemptive scheduling challenges.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scheduling of Real-Time GPU-Utilizing Tasks

In this dissertation, we tackle the challenges of GPU power management and analyzable guarantees for real-time GPU-using tasks. With the increasing performance capabilities of GPUs, there is a corresponding rise in power consumption, which poses challenges in terms of reliability, feasibility, and scalability. We explore innovative approaches to GPU power management that aim to optimize power consumption while ensuring reliable and efficient operation. Additionally, we address the challenges associated with analyzable guarantees in real-time systems where the interaction between CPUs and GPUs can be complex. The monopolization of resources by GPU tasks can result in delays for time-sensitive operations, impacting the overall system performance. Through comprehensive analysis and innovative techniques, we strive to provide analyzable guarantees and improve the predictability and responsiveness of real-time systems utilizing GPUs.

1.2 Dissertation Outlines and Contributions

Thesis Statement: This dissertation presents novel approaches to real-time GPU scheduling that address challenges in energy efficiency and preemptive scheduling, enabling optimized resource allocation, improved responsiveness, and enhanced schedulability in heterogeneous multi-GPU systems.

This statement is supported by the remaining chapters of this dissertation. We give a brief overview and contributions of each chapter in the following.

1.2.1 Chapter 2: sBEET: Balancing Energy Efficiency and Real-Time Performance in GPU Scheduling

Chapter 2 presents sBEET, a scheduling framework that Balances Energy Efficiency and Timeliness of GPU kernels 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.

Contributions[1]**.** This work 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.

[1]This work was published at RTSS 2021 [68]

- 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.

1.2.2 Chapter 3: sBEET-mg: Towards Energy-Efficient Real-Time Scheduling of Heterogeneous Multi-GPU Systems

In Chapter 3, we propose a multi-GPU real-time scheduling framework, sBEET-mg, that builds upon prior work on single-GPU systems to optimize scheduling strategies for heterogeneous multi-GPU systems. sBEET-mg 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.

Contributions[2]**.** This work makes the following contributions:

- 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.

[2]This work was published at RTSS 2022 [67]

1.2.3 Chapter 4: Unleashing the Power of Preemptive Priority-Based Scheduling for Real-Time GPU Tasks

In Chapter 4, we propose two novel techniques, namely the kernel thread and IOCTL-based approaches, to enable preemptive priority-based scheduling for real-time GPU tasks. Our approaches exert control over GPU context scheduling at the device driver level and enable preemptive GPU scheduling based on task priorities. The kernel thread-based approach achieves this without requiring modifications to user-level programs, while the IOCTL-based approach needs only a single macro at the boundaries of GPU access segments. In addition, we provide a comprehensive response time analysis that takes into account overlaps between different task segments, mitigating pessimism in worst-case estimates. Through empirical evaluations and case studies, we demonstrate the effectiveness of the proposed approaches in improving taskset schedulability and timeliness of real-time tasks. The results highlight significant improvements over prior work, with up to 40% higher schedulability, while also achieving predictable worst-case behavior on an Nvidia Jetson embedded platform.

Contributions[3]**.** This work makes the following contributions:

- We propose the kernel-thread and IOCTL-based approaches for preemptive priority-driven GPU scheduling in multi-core systems with an Nvidia GPU, providing implementation details and discussing their runtime characteristics.

- While it is important to run GPU segments according to their original task priority (esp. when task priority is assigned based on criticality), we find that assigning differ-

[3] As of the completion of this dissertation, this work is in the submission of RTSS 2023.

ent priorities to GPU segments can yield a significant benefit in taskset schedulability. Our work allows this.

- We present a comprehensive analysis on the worst-case task response time under our two proposed approaches. In particular, our analysis for the IOCTL-based approach considers self-suspension and busy-waiting modes during GPU kernel execution and reduces pessimism by taking into account the overlaps between different task execution segments.

- Our work is implemented on the latest Nvidia Tegra driver and will be open sourced. Experimental results show that our proposed approaches bring substantial benefits in taskset schedulability compared to previous synchronization-based approaches. A case study on Jetson Xavier NX demonstrates the effectiveness of our work over the default GPU driver.

Chapter 2

sBEET: Balancing Energy Efficiency and Real-Time Performance in GPU Scheduling

2.1 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 [50], 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.

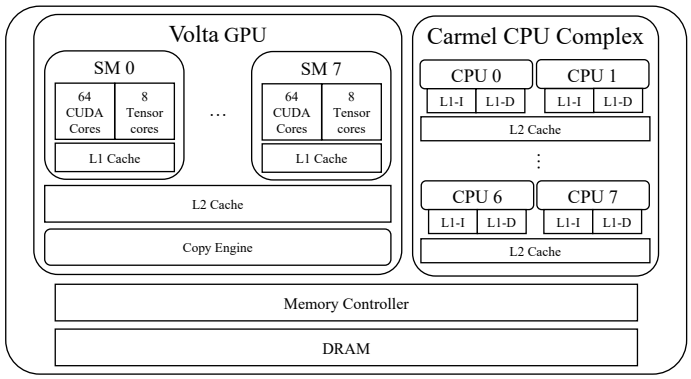
It is well known that each kernel may not be fully utilizing all the internal computing units of a GPU [8]. In order to better utilize the GPU resources and reduce the waiting time when multiple GPU kernels are sharing a GPU, recent real-time GPU scheduling schemes [38, 40, 58, 59] employ the *spatial 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 next sections, 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 work presents sBEET, a scheduling framework that Balances Energy Efficiency and Timeliness of GPU kernels to address the aforementioned challenges on embedded

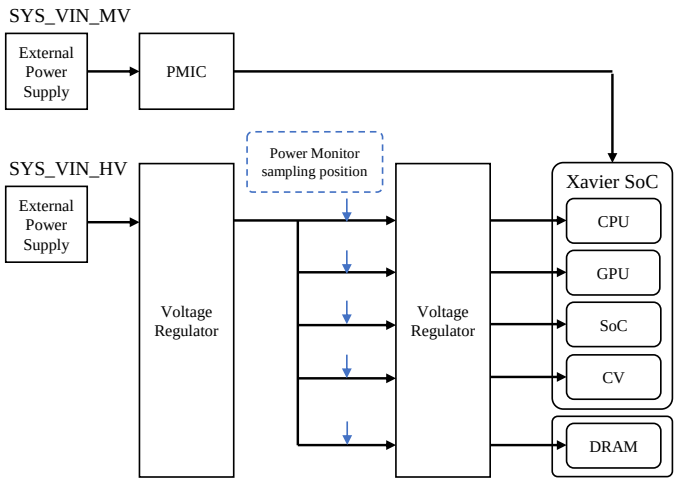
[1] Power-gating overhead is one of the major obstacles since each execution unit of a GPU tends to idle for shorter periods than break-even time [7].

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 [10], the fixed-priority Rate Monotonic (RM) scheduling without spatial multitasking, and the scheduling algorithm proposed in [58] 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 approach.



(a) Xavier SoC Architecture



(b) Block diagram of Xavier power rails

Figure 2.1: Architecture and module power rails of Nvidia Jetson AGX Xavier

2.2 Background and System Model

2.2.1 Background

Nvidia Jetson AGX Xavier. This work 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. 2.1a. 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 [10, 34, 45]. 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 [10, 52] 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. 2.1b 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 2.5.2, SM-level power/clock gating does not appear to exist even on the latest Xavier AGX GPU.

2.2.2 System Model and Assumptions

We consider a taskset Γ 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 τ_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 τ_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 τ_i, and is smaller than or equal to the period, i.e., $D_i \leqslant T_i$.

A task τ_i consists of a sequence of jobs $J_{i,j}$, where $J_{i,j}$ indicates the j-th job of task τ_i. Following the idea of spatial GPU multitasking [38, 40, 58, 59], each job $J_{i,j}$ of the task τ_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 τ_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 number of SMs on the device and $U_i(m) = \frac{G_i(m)}{T_i}$. It is worth noting that $U_i(M) \leqslant 1$; otherwise, τ_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 [59]:

- $J_{i,j}$ is a linear-speedup job if $G^e_{i,j}(m)$ is inversely proportional to m, i.e., $\forall m(m \leqslant M), G^e_{i,j}(m) = G^e_{i,j}(1)/m$. 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., $\exists m(m \leqslant M)$, $G^e_{i,j}(m) > G^e_{i,j}(1)/m$. 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.) [8].

This categorization will be used for proofs of energy consumption properties in spatial multitasking (Sec. 2.4.1), but our proposed runtime scheduling algorithm (Sec. 2.4.2) works regardless of the kernel type.

2.3 Related Work

Temporal Multitasking on GPU. Some prior works have been done to improve GPU utilization in a time domain. TimeGraph [43] is a real-time GPU task scheduler that assigns temporal budgets to tasks with different priorities, and replenishes the budgets periodically.

Elliott et al. [28] 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. [**?**] 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 [**?**, 28, 43] 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 [14, 42, 75] 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 [8]. 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 [6, 73]).

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 [8, 9, 38, 49, 61, 69]. Specifically, Jain et al. [38] 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. [59] 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. [40] 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. [71] 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. [58] 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 2.4.1, we will analyze the energy consumption of GPU tasks when spatial multitasking techniques are applied, recognizing

that such approaches may not necessarily result in the optimal energy-efficient schedule on the GPU.

Resource Allocation for GPU Energy Saving. Wang and Ranganathan [66] developed an instruction-level prediction mechanism to save energy by estimating the number of SMs for a given application. Wang et al. [65] 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 [32] 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. [9] presented QoS-aware dynamic resource allocation and experimentally demonstrated the effectiveness of this method in GPGPU-Sim. Sun et al. [60] 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. [74] 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. [62] 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 per-SM power-gating is not yet available in today's GPUs.

2.4 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.

2.4.1 Power and Energy Analysis

Power Model. Isci and Martonosi [36] 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 [32] and *GPUWattch* by Leng et al. [48]. 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:

$$P = P^s + P^d + P^{idle} \tag{2.1}$$

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 [32].[2] Hence, P^d can be decomposed into a linear sum of per-SM power consumed by each job. For a set of jobs J = $\{J_1, J_2, ..., J_n\}$[3] executing simultaneously on the GPU at time t, Eq. (3.1) can be rewritten as:

$$P = P^s + \sum_{i=1}^{n} P_i^d(m_i) + P^{idle}(M - \sum_{i=1}^{n} m_i) \tag{2.2}$$

[2]The dynamic power characteristics of each kernel can be estimated by either measurement-based profiling or analytical methods [32]. 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.

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 [32], the following conditions hold:

$$P_i^d(m) \propto m, \quad \text{and} \quad P^{idle}(m) \propto m \tag{2.3}$$

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

$$E = \int_{t_1}^{t_2} P dt \tag{2.4}$$

Now let us consider a set of jobs 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. (2.4) can be re-written as follows:

$$E(t_1, t_2) = \int_{t_1}^{t_2} \left(P^s + \sum_{i=1}^{n} \left(P_i^d \left(\sum_{k=1}^{M} x_i^k(t) \right) \right) \quad + P^{idle} \left(M - \sum_{i=1}^{n} \sum_{k=1}^{M} x_i^k(t) \right) \right) \tag{2.5}$$

The detailed methods to obtain the above power parameters and to realize SM allocation on commodity GPU hardware will be explained in Section 2.5.1. Based on the

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

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 $G_1^e(m_1) \leqslant G_2^e(m_2)$, 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 δ can be respectively written as below:

$$E_a = P^s \cdot \delta + P_1^d(M) \cdot G_1^e(M) + P_2^d(M) \cdot G_2^e(M) \tag{2.6}$$

$$\begin{aligned} E_b = {} & P^s \cdot \delta \\ & + (P_1^d(m_1) + P_2^d(m_2)) \cdot G_1^e(m_1) \\ & + (P_2^d(m_2) + P^{idle}(M - m_2)) \cdot (G_2^e(m_2) - G_1^e(m_1)) \end{aligned} \tag{2.7}$$

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,

$$\begin{aligned} & E_a - E_b > 0 \\ \Leftrightarrow \quad & G_1^e(1) \cdot (\frac{P_1^d(M)}{M} - \frac{P_1^d(m_1)}{m_1}) \\ & + G_2^e(1) \cdot (\frac{P_2^d(M)}{M} - \frac{P_2^d(m_2)}{m_2}) \\ & + P^{idle}(M - m_2) \cdot (G_1^e(m_1) - G_2^e(m_2)) \\ & > 0 \end{aligned} \tag{2.8}$$

From Eq. 2.3, 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. 2.8 to:

$$\begin{aligned} & E_a - E_b > 0 \\ \Leftrightarrow \quad & P^{idle}(M - m_2) \cdot (G_1^e(m_1) - G_2^e(m_2)) > 0 \end{aligned} \tag{2.9}$$

That leads to $G_1^e(m_1) > G_2^e(m_2)$. It contradicts to our assumption of $G_1^e(m_1) \leqslant G_2^e(m_2)$. 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, $\exists m (m \leqslant M)$, $G_{i,j}^e(m) > G_{i,j}^e(1)/m$. 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. 2.8. ■

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 3 *The energy-optimal number of SMs m_i^{opt} for a task τ_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 $\delta \geqslant \max_{m \leqslant M} G_{i,j}^e(m)$.*

The time interval δ considered for m_i^{opt} is to take into account the impact of idling SM time when τ_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 τ_i, because once τ_i completes execution, the GPU is power-gated and only the static power P^s contributes to the total energy consumption.

Example 4 *Consider a taskset Γ 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 2.1. 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 2.1: Taskset in Example 4

Task	D_i	$G_i^e(M)$	G_i^{hd}	G_i^{dh}	Offset
τ_1	12	6	1	1	0
τ_2	7	1	1	1	1

Fig. 2.2a shows the schedule without spatial multitasking. Since there are only two tasks and τ_1 arrives earlier than τ_2, any work-conserving scheduling policies would yield the

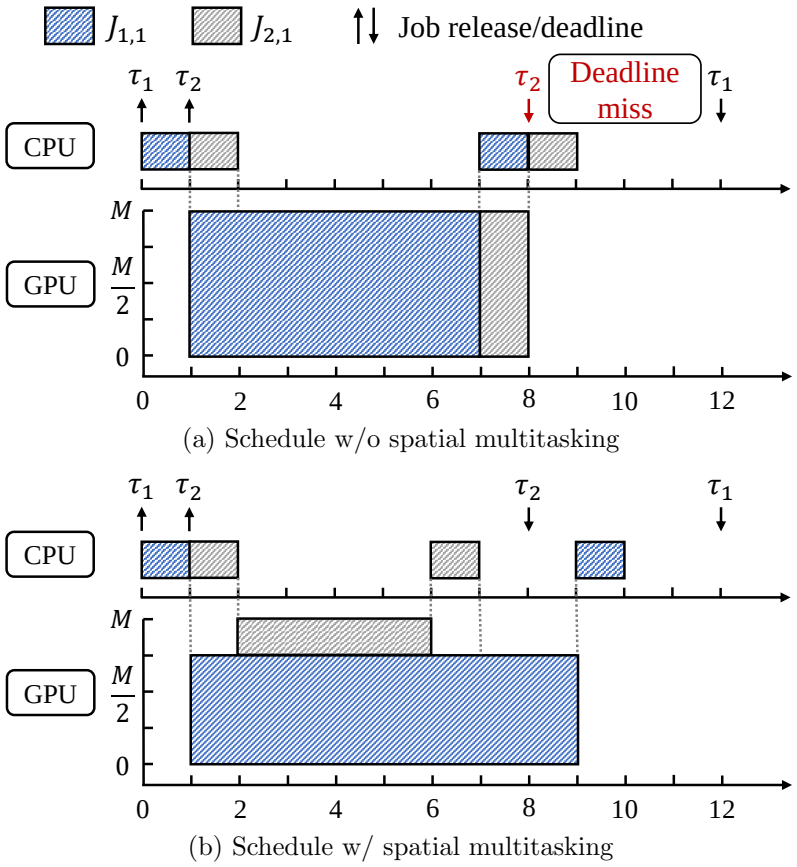


Figure 2.2: Scheduling results in Example 4

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. 2.2b 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. 2.2b is greater than that in Fig. 2.2a, 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 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.

2.4.2 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 2.4.1 (also see results in Fig. 2.10); (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 [8]; 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. 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.

Algorithm 1 Runtime Scheduler

Input: $J_{i,j}$: the first job in the ready queue

Input: S_{avail}: the set of currently available SMs

1: **procedure** SCHEDULER($J_{i,j}$, S_{avail})

2: **if** $|S_{avail}| = M$ **then** ▷ GPU is idling

3: $S_{i,j}^{cfg} \leftarrow$ ALLOCATION($J_{i,j}$, $nullptr$) ▷ Alg. 2

4: **if** $S_{i,j}^{cfg} \neq \varnothing$ **then**

5: Execute $J_{i,j}$ with $S_{i,j}^{cfg}$; remove $J_{i,j}$ from ready queue

6: **else if** $0 < |S_{avail}| < M$ **then**

7: $J_{q,r} \leftarrow$ currently running job

8: $S_{i,j}^{cfg} \leftarrow$ ALLOCATION($J_{i,j}$, $J_{q,r}$) ▷ Alg. 2

9: **if** $S_{i,j}^{cfg} \neq \varnothing$ **then**

10: Execute $J_{i,j}$ with $S_{i,j}^{cfg}$; remove $J_{i,j}$ from ready queue

11: **if** $S_{avail} = \varnothing$ or $S_{cfg} = \varnothing$ **then** ▷ $J_{i,j}$ not executed

12: Repeat the procedure for the next jobs in the ready queue until $S_{avail} = \varnothing$ or every job has been visited

(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 the information of $J_{q,r}$. If no valid SM allocation is found ($S_{i,j}^{cfg} = \varnothing$) or all SMs are busy ($S_{avail} = \varnothing$), 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. 2.4.1, 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:

Algorithm 2 SM Allocation

function ALLOCATION($J_{i,j}$, $J_{q,r}$)
  $t_{now} \leftarrow$ current time
  if $J_{q,r}$ is *nullptr* **then** ▷ $\Longrightarrow$ GPU is idling
    for $m \leftarrow M$ to 1 **do**
      $m' \leftarrow \min(m, m_i^{opt})$
      $Q_{i,j}^w \leftarrow \{J_{k,p} \mid \forall p, (\tau_k \neq \tau_q) \wedge (r_{k,p} < f_{i,j}(m'))\}$
      SCHEDGEN($J_{i,j}$, $J_{q,r}$, m', $[t_{now}, f_{i,j}(m')]$, $Q_{i,j}^w$)
      Compute $E_{pred} = E(t_{now}, f_{i,j}(m'))$ by Eq. (2.5)
    if no generated schedule is feasible **then**
      Choose the schedule with the minimum E_{pred}
    else
      Choose the feasible schedule with the min. E_{pred}
    return $S_{i,j}^{cfg}$ ▷ the corresponding SM allocation for $J_{i,j}$
  else ▷ the GPU is partially occupied
    $m' \leftarrow \min(|S_{avail}|, m_i^{opt})$
    if $f_{i,j}(m') > f_{q,r} + G_{i,j}^e(M)$ **then**
      return $\varnothing$ ▷ Do not run $J_{i,j}$ in parallel with $J_{q,r}$
    else
      $Q_{i,j}^w \leftarrow \{J_{k,p} \mid \forall p, (\tau_k \neq \tau_q) \wedge (r_{k,p} < f_{i,j}(m'))\}$
      SCHEDGEN($J_{i,j}$, $J_{q,r}$, m', $[t_{now}, f_{i,j}(m')]$, $Q_{i,j}^w$)
      if the generated schedule is not feasible **then**
        return $\varnothing$
      else
        return $S_{i,j}^{cfg}$ ▷ the corresp. SM allocation

Algorithm 3 Schedule Generation

1: **function** SchedGen($J_{i,j}$, $J_{q,r}$, m', $[t_{now}, t_{fin}]$, $Q^w_{i,j}$)
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 $J_{k,p} \in Q^w_{i,j}$
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: /* Consider other jobs in $Q^w_{i,j}$ */
10: $S_{rem} \leftarrow$ # of remaining (unused) SMs at $t_{k,p}$
11: **for** $J_{k,p} \in Q^w_{i,j}$ in ascending order of arrival time **do**
12: $m'' \leftarrow \min(S_{rem}, m^{opt}_k)$
13: **if** $m'' = 0$ **then**
14: **continue** ▷ Ignore this job from parallel exec.
15: Place $J_{k,p}$ with m'' SMs at t_{next}
16: $t_{next} \leftarrow f_{k,p}(m'')$
17: **if** $t_{next} \geqslant t_{fin}$ **then**
18: **break** ▷ Stop schedule generation
▷ Schedule generation done

- (Alg. 2 line 3 to 13) If the GPU is idling, the algorithm iterates through m from M to 1. For each m, it assigns $m' = \min(m, m^{opt}_i)$ 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 $Q^w_{i,j}$ in an ascending order of arriving time. Then the algorithm calls the SchedGen function in Alg. 3 (explained below) to generate a schedule of $J_{i,j} \cup Q^w_{i,j}$ for a time interval

$[t_{now}, f_{i,j}(m')]$. The algorithm predicts the energy consumption E_{pred} of the generated schedule by Eq. 2.5 (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 9 to 12). Finally, the algorithm returns the corresponding SM configuration $S_{i,j}^{cfg}$ for $J_{i,j}$ (line 13).

- (Alg. 2 line 14 to 24) 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 $\varnothing$ (line 17) 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 18 to 24).

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 exists) in the schedule (line 3 to 8). 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 10 to 18). 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 13 to 14).

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

Example 5 *Consider a taskset* Γ *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 2.2. Fig. 2.3a 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.*

Table 2.2: Taskset in Example 5

Task	D_i	$G_i^e(M)$	G_i^{hd}	G_i^{dh}	Offset
τ_1	14	6	1	1	0
τ_2	7	1	1	1	1
τ_3	10	1	1	1	2

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. 2.3a 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. 2.3b where* $\frac{3 \cdot M}{4}$ *SMs is assigned to* $J_{1,1}$*. In*

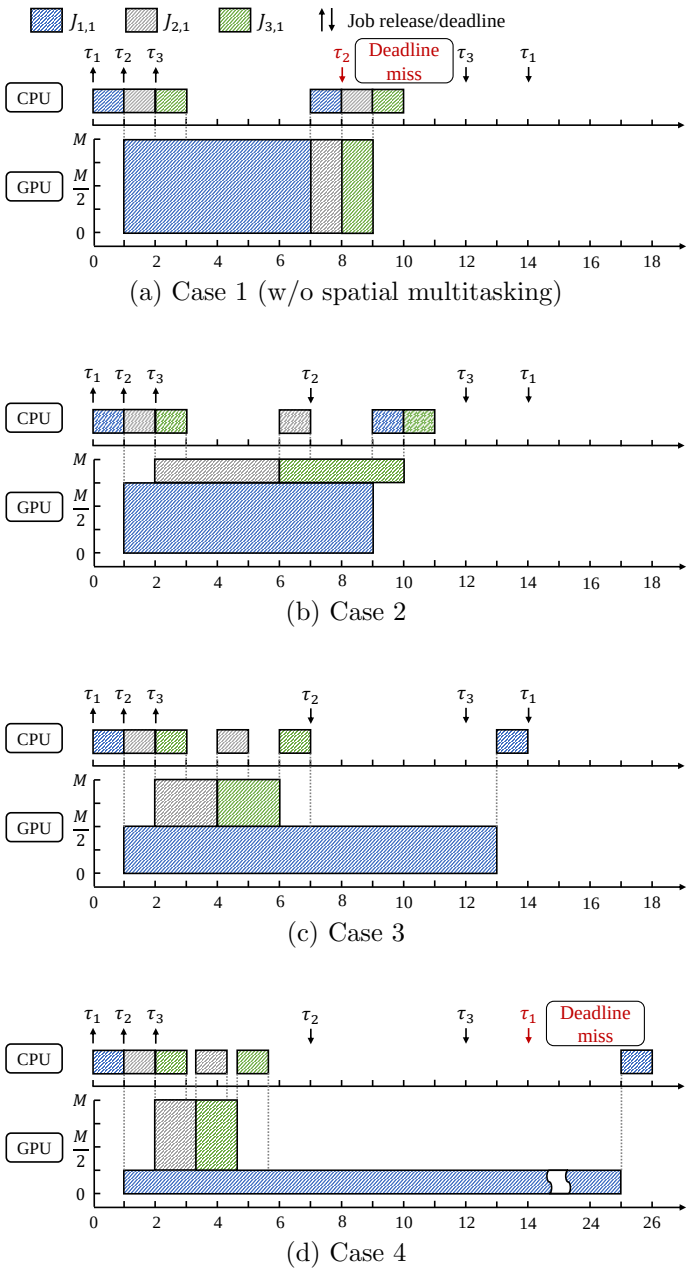


Figure 2.3: Scheduling results in Example 3

this case, since fewer SMs are given to $J_{1,1}$, an interval of $G^e_{1,1}(\frac{3 \cdot M}{4})$ 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. 2.3c and Fig. 2.3d. The schedule in Fig. 2.3d 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. 2.3b and 2.3c. Hence, the scheduler selects the schedule in Fig. 2.3b since it has a lower energy consumption computed by the approach in Theorem. 1.

(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 11 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.

2.5 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.

2.5.1 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. 2.3 holds [32]; 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. 2.2.

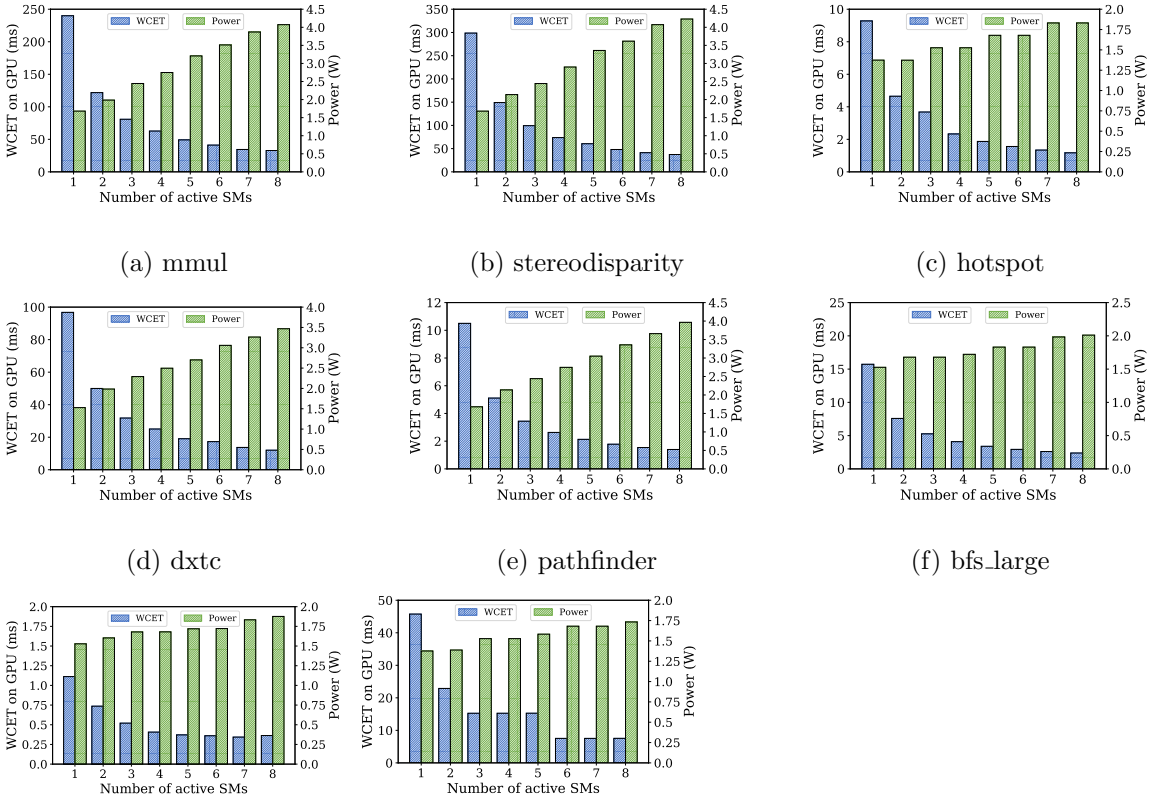
Benchmarks Selection. In the evaluation, We consider eight different GPU benchmarks whose execution time is not too short ($> 100\mu s$) so that the overlapped kernel execution

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 [25], 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 [26, 39, 58] 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.

2.5.2 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



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

(d) dxtc (e) pathfinder (f) bfs_large

(g) bfs_small (h) synthetic kernel

Figure 2.4: Profiling results of WCET and power consumption

as the WCET. Fig. 2.4 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.

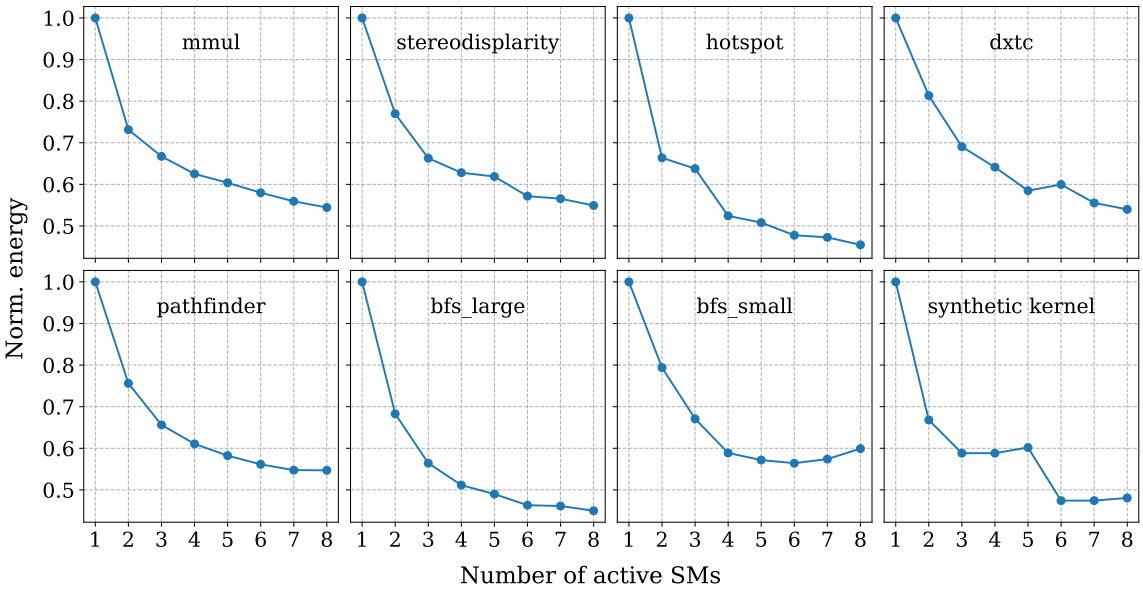


Figure 2.5: Normalized energy consumption in time window

2.5.3 Energy Consumption in an Observation Window

To find out m_i^{opt} in Def. 10, we have observed the energy consumption of each benchmark with a different number of SMs, as shown in Fig. 2.5. 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. 2.5 using the profiling results in Section 2.5.2. 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.

2.5.4 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 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. 2.6 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 [21] 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.

2.5.5 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 [58] that

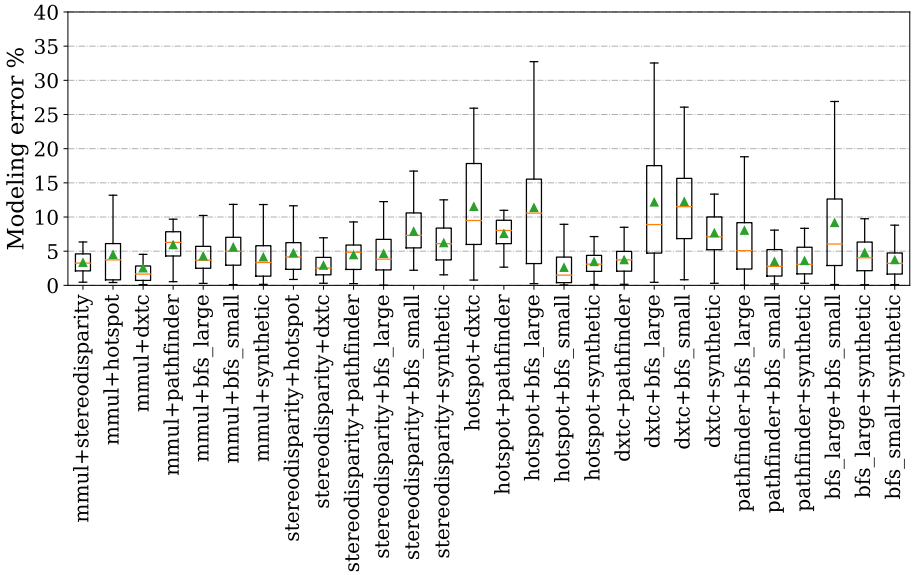
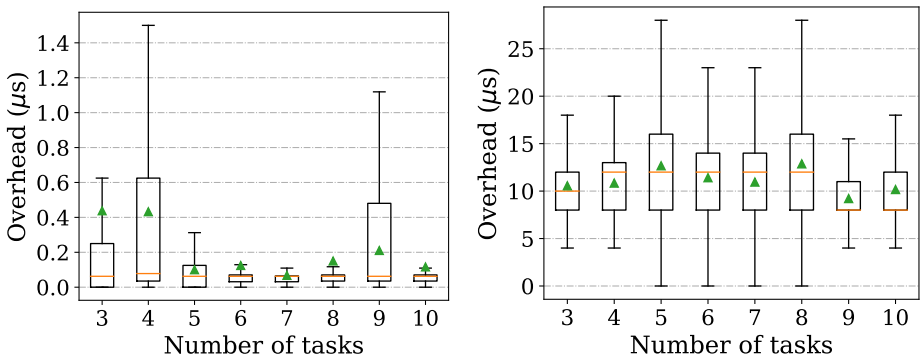


Figure 2.6: Error of predicted GPU power consumption

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.



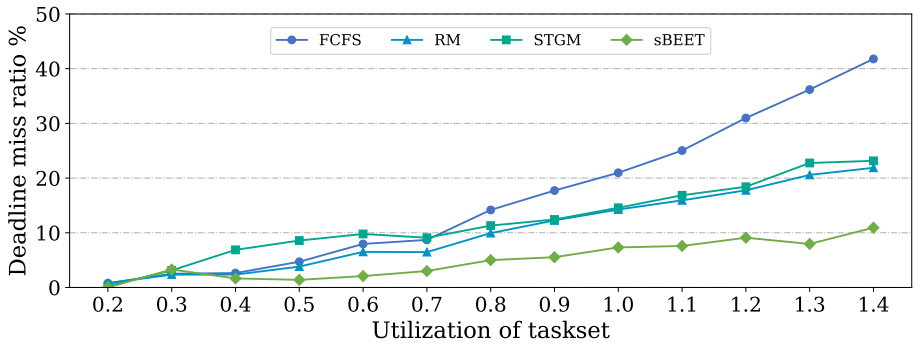
(a) Overhead of Alg. 1

(b) Overhead of Alg. 2 and 3

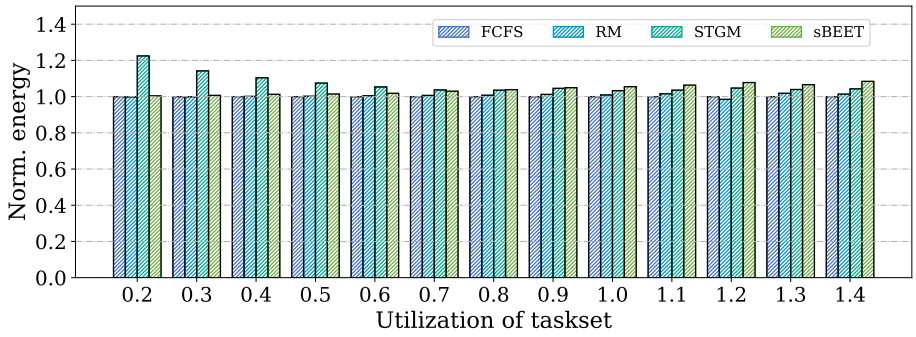
Figure 2.7: Runtime overhead w.r.t number of tasks

Overhead Measurement

We measured the runtime overhead of sBEET and the results are shown in Fig. 2.7. 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 2.5.1. To obtain the overheads of the proposed runtime algorithms in Section 2.4.2 , 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 μs, we conclude that the sBEET framework is suitable to use on embedded platforms at runtime.



(a) Deadline miss ratio



(b) Overall energy consumption

Figure 2.8: Runtime results w.r.t. the utilization of taskset

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. 2.2.2 and determined by the UUniFast algorithm [16]), 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 [28, 45, 56]. For STGM, eight workers are created since STGM does not limit the number of jobs that can run simultaneously on the GPU.

In Fig. 2.8, we show the performance of the four scheduling approaches for various utilization settings. Fig. 2.8a presents the deadline miss ratio under the four approaches, and sBEET has the lowest deadline miss ratio among them. Note that, when $U \leqslant 0.7$, 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 $U \geqslant 0.8$, FCFS becomes the worst among the four approaches. The curves of STGM and RM overlap due to the fallback mentioned in Section 2.5.5.

Fig. 2.8b shows the runtime energy consumption of the four approaches, normalized to the case of FCFS. We observe that, when $U \leqslant 0.7$, 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 2.4.1. Another reason is that FCFS, RM and STGM have higher deadline miss ratios, as can be seen in Fig. 2.8a. In other words, during the same observation interval, they have fewer completed jobs compared to sBEET.

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 [46], 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. 2.9 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. 2.2. Therefore, sBEET has a better performance in meeting the deadlines of light tasks than the other approaches.

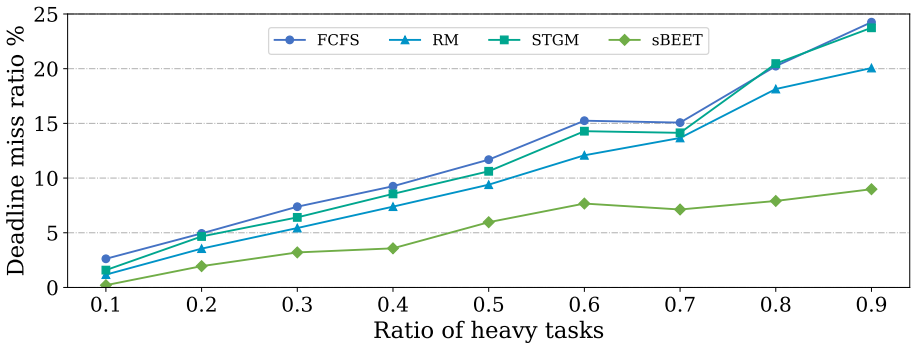


Figure 2.9: Runtime deadline miss ratio of light tasks w.r.t. ratio of heavy tasks

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 [58]), 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. The measurement results of runtime energy consumption are shown in Fig. 2.10. 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 2.4.1, and also supports the reasoning that having more workers may not help improve energy consumption and schedulability.

2.5.6 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

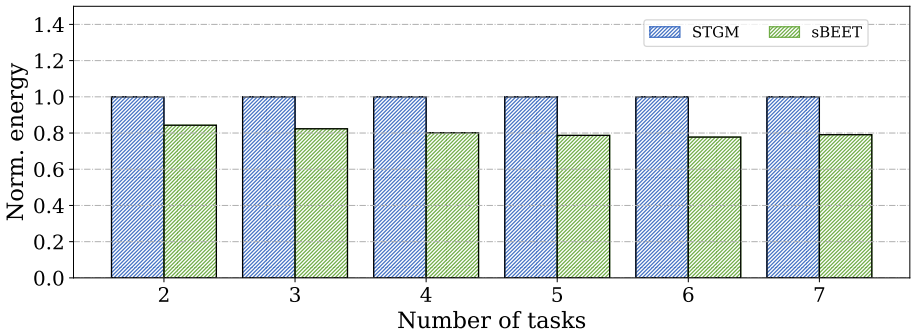


Figure 2.10: Comparison of runtime energy consumption of STGM and the proposed work

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 [38], 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.

2.6 Conclusion

In this work, 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.

Chapter 3

sBEET-mg: Towards Energy-Efficient Real-Time Scheduling of Heterogeneous Multi-GPU Systems

3.1 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 [50], 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 [33, 35, 47]. 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 [8], the idle energy consumption of the computing units of an active GPU causes energy inefficiency [37, 68] 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 work 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 [68] 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.

3.2 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 [28–30, 45, 56]. In particular, Elliott et al. [30] 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* [9, 49, 61]. Recent studies have shown that spatial multitasking offers better performance isolation and concurrency [38] and better schedulability and resource utilization [58, 76] 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 [9, 60, 62, 65], based on an assumption that the unused SMs can be turned off and energy consumption can be reduced in the presence of SM-level power-gating. For example, Hong and Kim [65] focused on finding the optimal number of SMs for the highest performance-per-Watt. Aguilera et al. [9] and Sun et al. [60] 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. The unused SMs by those methods consumes active-idle power as long as the GPU is not fully idle. The incapability to power-gate individual SMs also makes the energy management problem of GPUs different from that of multi-core CPUs.

Recently, Wang et al. [68] proposed an energy-efficient real-time GPU scheduler, called sBEET. They first showed that although spatial multitasking benefits schedulability,

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

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.

3.3 Background and System Model

3.3.1 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 [10, 38, 44, 53, 55, 68].

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 [38, 40, 58, 59] 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 [21, 65], our experiments have confirmed that it is still not available on Nvidia's latest Ampere architecture. This matches with the observations of the recent paper [68].

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.

3.3.2 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 3.1.

[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.

Platform Model. We consider a single-ISA system Π consisting of ω heterogeneous GPUs. The k-th GPU in the system is denoted by π_k, and each GPU is characterized by its power model, computational capacity and clock speed. The GPU π_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 π_k, e.g., $type(\pi_k) = type(\pi'_k)$ means two GPUs are identical.

Task Model. We consider a taskset Γ 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 τ_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 τ_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 τ_i, and is smaller than or equal to the period, i.e., $D_i \leqslant T_i$.

A task τ_i consists of a sequence of jobs $J_{i,j}$, where $J_{i,j}$ indicates the j-th job of task τ_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 [38, 40, 58, 59], each job $J_{i,j}$ of the task τ_i can execute with a different number of SMs on a different GPU. Hence, we use $G_{i,j}(m, \pi_k)$

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

to represent the WCET of $J_{i,j}$, where m denotes the number of SMs used by $J_{i,j}$ on the GPU π_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 π_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 π_k
- $G_i^{dh}(\pi_k)$: the worse-case data copy time from the device to the host memory on the GPU π_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 τ_i on a GPU π_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 π_k. The utilization of τ_i with m SMs on π_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 π_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.

Power Model. Following the power modeling approach in [32, 36, 68], the power consumption of a GPU at time t can be represented as follows:

$$P = P^s + P^d + P^{idle} \tag{3.1}$$

Table 3.1: Symbols and their definitions in this work

Notation	Definition
π_k	The k-th GPU in the system
M_k	The total number of SMs on the GPU π_k
M_k^{limit}	The number of SMs that allowed (by the user) on the π_k
G_i	The cumulative WCET of GPU segments of task τ_i
T_i	The period of task τ_i
D_i	The relative deadline of task τ_i, and $D_i \leqslant T_i$
$J_{i,j}$	The j-th job of task τ_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 τ_i on π_k
$G_i^{dh}(\pi_k)$	The WCET of device to host memory copy of τ_i on π_k
$G_i^e(m, \pi_k)$	The WCET of kernel execution of τ_i on π_k with m SMs
$U_i(\pi_k)$	The utilization of task τ_i on π_k
$U(\pi_k)$	The utilization of π_k

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 [32]. 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 π_k at time t, the power consumption of the GPU π_k, P_k, can be computed as follows:

$$P_k = \begin{cases} P_k^s + \sum\limits_{J_i \in \mathrm{J}} P_{k,i}^d(m_i) + P_k^{idle}(M_k - \sum\limits_{J_i \in \mathrm{J}} m_i) & \text{if } \mathrm{J} = \varnothing \\ P_k^s & \text{if } \mathrm{J} \neq \varnothing \end{cases} \tag{3.2}$$

where $m_1, m_2, ...$ are the number of SMs that are being used by $J_1, J_2, ...$ at time t ($\sum m_i \leqslant M_k$). $P_{k,i}^d(m_i)$ is the dynamic power consumption of J_i on π_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 π_k. Since dynamic and idle power is known to be linear to the number of SMs [32], $P^d_{k,i}(m_i) = m_i \cdot P^d_{k,i}(1)$ and $P^{idle}_k(m) = m_i \cdot P^{idle}_k(1)$ 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^d_k and P^{idle}_k, i.e. $\sum P^d_k(0) = 0$ and $P^{idle}_k(M_k) = 0$.

In this work, we directly measured these power parameters of using our test-bed setup (Sec. 3.4.1), but they can also be estimated using analytical methods [32].

Energy Consumption. We adopt the energy computation method in Eq. 5 in [68]. Let us consider a set of jobs J = $\{J_1, J_2, ...\}$ that are *scheduled* on the GPU π_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 $x^m_i(t)$ 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 π_k can be computed by:

$$\begin{aligned} E_k([t_1, t_2]) = \int_{t_1}^{t_2} \Bigg(P^s_k + \sum_{J_i \in \mathrm{J}} \Big(P^d_{k,i}(\sum_{m=1}^{M_k} x^m_i(t)) \Big) \\ + P^{idle}_k \Big(M_k - \sum_{J_i \in \mathrm{J}} \sum_{m=1}^{M_k} x^m_i(t) \Big) \Bigg) dt \end{aligned} \tag{3.3}$$

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

$$E([t_1, t_2]) = \sum_{\forall \pi_k \in \Pi} E_k([t_1, t_2]) \tag{3.4}$$

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.

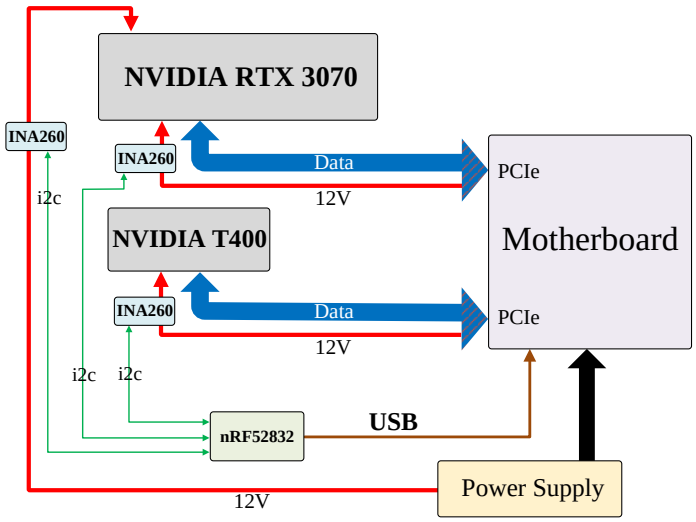
3.4 Energy Usage Characteristics of Multi-GPU Systems

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.

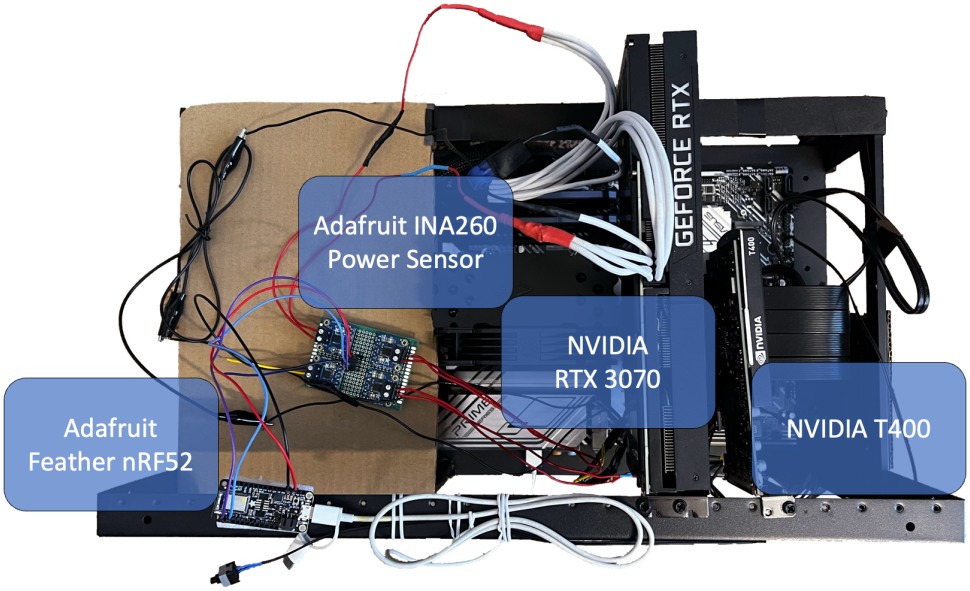
3.4.1 Hardware Setup

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 power sensor, its accuracy is not high ("+/- 5 watts" according to the official document [5]) and the power reading is not available on the T400 device. The



(a) Block diagram of our hardware setup



(b) Implementation of the block diagram

Figure 3.1: Multi-GPU system with a power monitoring tool

power measurements by the built-in sensor may show anomalies and need to be corrected as the prior work suggested [23].

Due to these reasons, we developed a custom hardware tool to obtain a precise measurement of power consumption by each GPU. Fig. 3.1 shows our system with two GPUs connected to the power monitoring tool. We used an INA260 sensor [63] 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 [51] 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 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 [41] presented in 2022 RTSS@Work.

3.4.2 Benchmarks and Power Profiles

Six benchmark programs are considered in our experiments: MatrixMul, Stereodisparity, DXTC, Histogram from Nvidia CUDA 11.6 Samples [3], and Hotspot, BFS from the Rodinia GPU benchmark suite [25]. 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. 3.2 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 3.2 shows the dynamic power parameters of the benchmarks and the idle and static power of the two GPUs. π_0 is RTX 3070 and π_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. 3.3.2. Interestingly, RTX 3070 does not always consume more

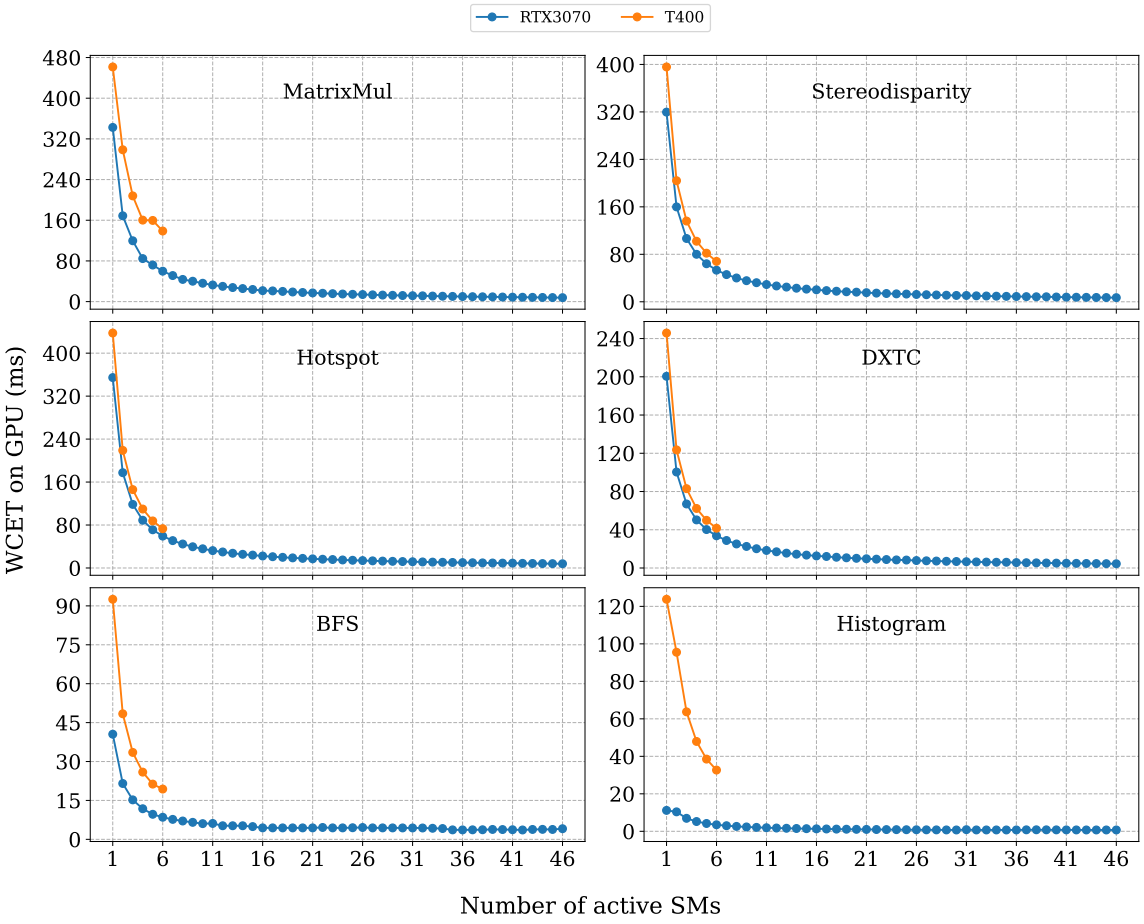


Figure 3.2: WCET of benchmarks on RTX 3070 and T400

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.

Table 3.2: 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
π_0 (RTX 3070)	46 W	0.445 W
π_1 (T400)	8 W	0.652 W

3.4.3 Observations

Using real execution time and power parameters, we provide examples to gain insights and make observations for 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 [30].

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. 3.2 and summarized in Tables 3.3 and 3.4. For ease of presentation, we focus on kernel execution time, G_i^e, and omit data copy time.

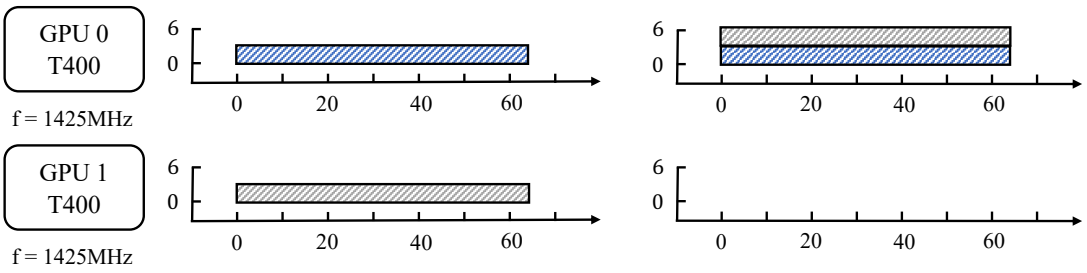
Homogeneous GPUs. Consider a homogeneous multi-GPU system $\Pi = \{\pi_0, \pi_1\}$ containing two identical Nvidia T400 GPUs, i.e., $type(\pi_0) = type(\pi_1)$.

Example 6 *Consider two tasks with the execution time parameters given in Table. 3.3. 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 lightly loaded system, we only enable 3 SMs on T400. We select an observation window of 100ms for the following two possible schedules shown in Fig. 3.3: schedules by load distribution and by load concentration. In Fig.3.3a, the job of τ_1, $J_{1,1}$, and the job of τ_2, $J_{2,1}$, are distributed to two GPUs, and the estimated energy consumption of this schedule is 2.3J computed by Eq. (3.4)*. Fig.3.3b shows the schedule under load concentration strategy. In this schedule, $J_{1,1}$ and $J_{2,1}$ share the GPU π_0 while leaving π_1 idle so that it can be power gated. The estimated energy consumption of the system is 2.05J.*

Table 3.3: Taskset in Examples 6 and 7

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



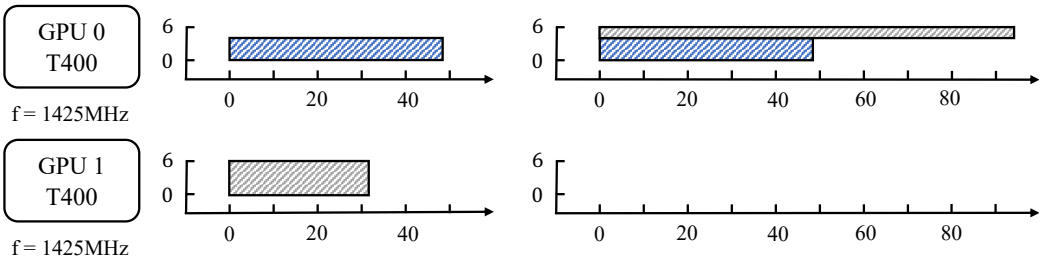
(a) Schedule w/ distributed load: E=2.3J

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

Figure 3.3: Scheduling results in Example 6

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 [68], 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 7 *Consider the same tasks as in Examples 6. Now, the job of* τ_1*,* $J_{1,1}$*, executes on* π_0 *with 4 SMs, and the execution time parameters are given in Table 3.3. In the schedule shown in Fig. 3.4a,* $J_{1,1}$ *and* $J_{2,1}$ *are distributed to* π_0 *and* π_1*, and* $J_{2,1}$ *executes on* π_1 *with 6 SMs. In the schedule in Fig. 3.4b,* $J_{2,1}$ *is assigned to* π_0 *with the remaining SMs and executes*



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

Figure 3.4: Scheduling results in Example 7

in with $J_{1,1}$ concurrently, while π_1 stays idle. However, after $J_{1,1}$ finishes execution, $J_{2,1}$ is still running, during which the idle SMs on π_0 keep consuming energy, and this makes it less energy efficient than the schedule with load distribution. With Eq. (3.4)*, 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. 3.4.1.

Example 8 *Consider a taskset with parameters listed in Table 3.4. Suppose at time $t = 0$, the job $J_{1,1}$ of τ_i, has just started its kernel execution on the GPU π_0 with 16 SMs, and at the same time, the job of τ_2, $J_{2,1}$, is ready for execution. By employing the load distribution approach, $J_{2,1}$ will execute on the GPU π_1 and the resulting schedule of the two tasks is shown in Fig. 3.5a. Similar to the previous examples, when an observation window of 100ms is considered, the estimated energy consumption of this schedule is calculated to be 7.35J.*

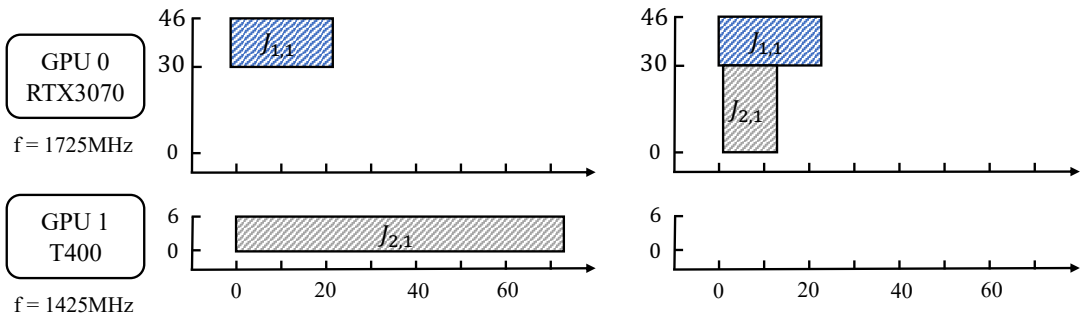
Table 3.4: Taskset in Example 8 and 9

Task	Application	$G_i^e(30, \pi_0)$	$G_i^e(16, \pi_0)$	$G_i^e(6, \pi_1)$
τ_1	MatrixMul	11.98 ms	21.55 ms	-
τ_2	Hotspot	12.00 ms	22.31 ms	73.188 ms

Fig. 3.5b shows the schedule under the load concentration approach. Since $J_{1,1}$ is not using all the SMs of the GPU π_0, $J_{2,1}$ is able to use the remainder. In this way, π_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 9 *Consider the same multi-GPU system and task parameters as in Example 8. But at this time, $J_{1,1}$ starts kernel execution with 30 SMs on π_k. Following the load concentration approach, $J_{2,1}$ uses the remaining 16 SMs on π_k as shown in Fig. 3.6b and the estimated energy consumption of this schedule is 7.3J. Since π_1 is idle when $J_{2,1}$ is ready for its execution, the load distribution approach executes $J_{2,1}$ on π_2 with all the available SMs. Fig. 3.6a shows this schedule and the estimated energy consumption here is 7.19J, which is smaller than that with the load concentration approach.*

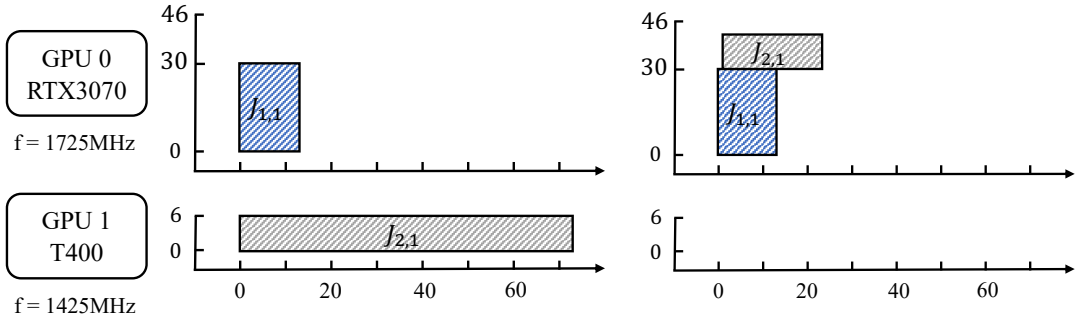
To summarize, the above examples suggest that neither load concentration nor distribution should be preferred over the other when making scheduling decisions in a multi-GPU system, regardless of whether GPUs are homogeneous or not. One thing we can clearly observe is that, if all tasks assigned to the 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



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

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

Figure 3.5: Scheduling results in Example 8



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

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

Figure 3.6: Scheduling results in Example 9

absolute completion time is determined only at runtime. The difficulty of this problem multiplies when timing constraints are considered.

3.5 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 [68], which is the latest work on energy-efficient real-time GPU scheduling for a single GPU system. Hence, we name our framework as "sBEET-mg".

3.5.1 Energy Optimality

To better explain the proposed scheduling framework, here we revisit the definition of the *energy-optimal* number of SMs given in [68] and give the definition of *energy-preferred* GPU for each task in a multi-GPU system.

Definition 10 (Energy optimal SMs [68]) *The energy-optimal number of SMs $m_{k,i}^{opt}$ for a task τ_i on a GPU π_k is defined as the number of SMs that leads to the lowest energy consumption computed by Eq.* (3.3) *when it executes in isolation on the GPU π_k during an arbitrary time interval $\delta \geqslant \max_{m \leqslant M_k} G_{i,j}^e(m, \pi_k)$.*

In the above definition, it is worth noting that the energy-optimal number of SMs is unaffected by the duration of δ. This is derived from Eq. (3.3). Assume the minimum possible $\delta_{min} = \max_{m \leqslant Mk} G_i^e, j(m, \pi_k)$, which is long enough for τ_i to complete execution no matter how many SMs are allocated. After δ_{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. (3.3), we can define the energy-preferred GPU as below.

Definition 11 *(Energy preferred GPU) The energy-preferred GPU for a task τ_i in a multi-GPU system Π is given by:*

$$\underset{\pi_k \in \Pi}{\operatorname{argmin}} \int_0^{\delta} P_k^s + P_{k,i}^d(m_{k,i}^{opt}) + P_k^{idle}(M_k - m_{k,i}^{opt})dt \tag{3.5}$$

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

3.5.2 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 closely resembles 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 server's scheduling algorithm determines the target GPU for dispatching a job, it forwards the job to the corresponding worker thread responsible for executing it on the designated GPU. The worker thread, in turn, leverages the cudaSetDevice() function to specify the GPU device and initiates the kernel in a separate CUDA stream. This design enables the availability of independent execution instances for each running job, thereby allowing for simultaneous utilization of multiple GPUs. Notably, GPU partitioning is achieved through the use of persistent threads, enabling parallel kernel execution across all GPUs within the system. The decision regarding which GPU to utilize and when to employ spatial multitasking is made by our subsequent scheduling algorithm, which will be discussed later in this work.

When the sBEET-mg framework starts, the procedure in Alg. 4 is executed to allocate tasks to GPUs. More details on this procedure will be explained below. Following the observation in [68], we limit the number of worker threads on each GPU to two since more parallelism does not necessarily improve performance [68]. 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. 5 (explained later) to decide whether to execute this job on the preassigned GPU by Alg. 4 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.

3.5.3 Offline Task Distribution

For a given taskset Γ, 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 π_x with $m_{x,i}^{opt}$ as long as the capacity of π_x permits. Alg. 4 depicts the pseudocode of the task distribution procedure. It first sorts all tasks in Γ in decreasing order of priority so that higher-priority tasks have a better chance to get their energy-preferred GPUs (line 2). Then for each task τ_i, it obtains a list Π_i of GPUs in non-increasing order

Algorithm 4 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:         if τ_i is not assigned then
10:            Assign τ_i to the GPU that has a minimum utilization after τ_i is assigned
```

of expected energy consumption. Hence, the energy-optimal GPU of τ_i goes first in this list. For each GPU π_k in the ordered list Π_i, it runs a simple utilization check to decide whether τ_i can be accepted (line 3 to line 8). After iterating through all the GPUs, if τ_i is still not assigned to any GPU, the algorithm assigns it to the GPU that will have the minimum utilization after τ_i is assigned (line 10). The result of this allocation serves as a guideline for the runtime scheduler.

3.5.4 Runtime Job Migration

Alg. 4 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 [37], 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 Chapter 2):

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

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

- π_k is the GPU that the caller (the runtime scheduler of sBEET-mg, namely Alg. 4) 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 $S_{i,j}^{cfg}$ and E. $S_{i,j}^{cfg}$ is the SM allocation result on π_k for $J_{i,j}$. If $S_{i,j}^{cfg} = \varnothing$, $J_{i,j}$ cannot execute on π_k for now. E is the expected energy consumption of Π during a time window from the current time to the estimated finish time of $J_{i,j}$, $f_{i,j}$, with the SM allocation $S_{i,j}^{cfg}$. Hence, by Eq. (3.4), E is equal to $E([t_{now}, f_{i,j}])$. If $S_{i,j}^{cfg} = \varnothing$, $E = \infty$.

Alg. 5 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 π_x. As a result of

scheduling decision making, the algorithm returns a SM configuration $S_{i,j}^{cfg}(\pi_k)$ for $J_{i,j}$. If $S_{i,j}^{cfg}(\pi_k) = \varnothing$, $J_{i,j}$ is pushed to the pending queue for later consideration.

- (Alg. 5 line 2 to 20) If π_x is idle, the scheduler tentatively puts $J_{i,j}$ on π_x with $m_{x,i}^{opt}$ 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 ∞, meaning the assignment is invalid (line 2 to 8). 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 π_x can be power gated to save energy. It iterates through Π_i which is obtained in Alg. 4, 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 $S_{i,j}^{cfg}$ with the smaller predicted energy consumption (line 9 to 20).

- (Alg. 5 line 21 to 23) In the second case, π_x is partially occupied. The scheduler calls sBEET(π_k, $J_{i,j}$) to decide and return the SM configuration $S_{i,j}^{cfg}$.

- (Alg. 5 line 24 to 38) If π_x is fully occupied, we consider the following two cases: (i) $J_{i,j}$ can be postponed and wait for $m_{x,i}^{opt}$ SMs on GPU π_x (line 25 to 31), or (ii) execute on a GPU other than π_x (line 32 to 36). In case (i), the scheduler estimates the time when π_x would become available with $m_{x,i}^{opt}$ 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

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

is set to ∞. In case (ii), the scheduler iterates through Π_i and predicts the energy consumption if $J_{i,j}$ is placed on a GPU other than π_x. For each $\pi_{k \neq x} \in \Pi_i$, if π_k is not fully occupied, the scheduler calls sBEET(π_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 $S_{i,j}^{cfg}$ of (i) or (ii) that leads to smaller energy consumption of $J_{i,j}$'s execution.

3.5.5 Time Complexity

According to the time complexity analysis in Chapter 2, 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 ω. In Alg. 5, the procedure to check whether a job can be scheduled on each GPU (lines 10 to 15 and 33 to 36) is upper-bounded by $\omega \cdot O(n \cdot log(n))$; hence, the time complexity of the runtime job migration is given by $O(\omega \cdot n \cdot log(n))$.

3.5.6 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.

Algorithm 5 Runtime Job Migration

function JOB MIGRATION($J_{i,j}$)
  if π_x is idle **then**
    Tentatively place $J_{i,j}$ on π_x with $m_{x,i}^{opt}$ SMs
    if $J_{i,j} \cup \pi_x$ will meet deadlines **then**
      $E_1 \leftarrow E([t_{now}, f_{i,j}])$
      $S_1 \leftarrow$ the corresponding SM allocation
    else
      $E_1 \leftarrow \infty$
    $E_2 \leftarrow \infty$
    for each $\pi_{k \neq x}$ in Π_i sorted by Alg. 4 **do**
      if π_k is idle or π_k is fully occupied **then**
        continue
      else
        $(S_2', E_2') \leftarrow$ sBEET(π_k, $J_{i,j}$)
        $E_2 \leftarrow \min(E_2, E_2')$
    if $E_1 == \infty$ and $E_2 == \infty$ **then**
      Assign $J_{i,j}$ with maximum SMs on π_x
    else
      Select the schedule with $\min(E_1, E_2)$
    return $S_{i,j}^{cfg}$ ▷ the corresponding SM allocation for $J_{i,j}$
  else if π_x is partially occupied **then**
    $(S_{i,j}^{cfg}, E) \leftarrow$ sBEET(π_x, $J_{i,j}$)
    return $S_{i,j}^{cfg}$

else ▷ If the GPU is full

  $t_1 \leftarrow$ current time

  Tentatively place $J_{i,j}$ on π_x and wait until $m_{x,i}^{opt}$ SMs become available

  $t_2 \leftarrow f_{i,j}$

  if $J_{i,j} \cup \pi_x$ will meet deadlines **then**

    $E_4 \leftarrow E([t_1, t_2])$

  else

    $E_4 \leftarrow \infty$

  $E_5 \leftarrow \infty$

  for each $\pi_{k \neq x}$ in Π_i sorted by Alg. 4 **do**

    if π_k is not full **then**

      $(S'_5,\ E'_5) \leftarrow \text{sBEET}(\pi_x,\ J_{i,j})$

      $E_5 \leftarrow \min(E_5, E'_5)$

  Select the schedule with $\min(E_4, E_5)$

  return $S_{i,j}^{cfg}$ ▷ the corresponding SM allocation for $J_{i,j}$

3.6 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. 3.4.1. To evaluate performance in systems with more GPUs, we also present experimental results from a Python simulator we developed (Sec. 3.6.2). 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"

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

(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. 5).

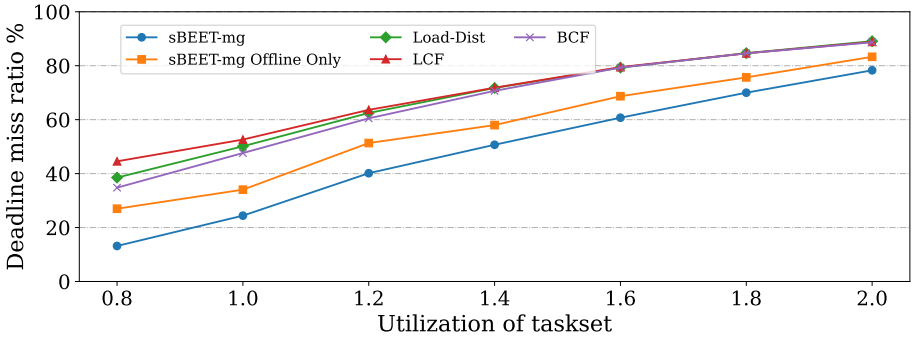
3.6.1 Hardware Experiments

In all the experiments on real hardware, we use the system shown in Fig. 3.1 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 π_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.

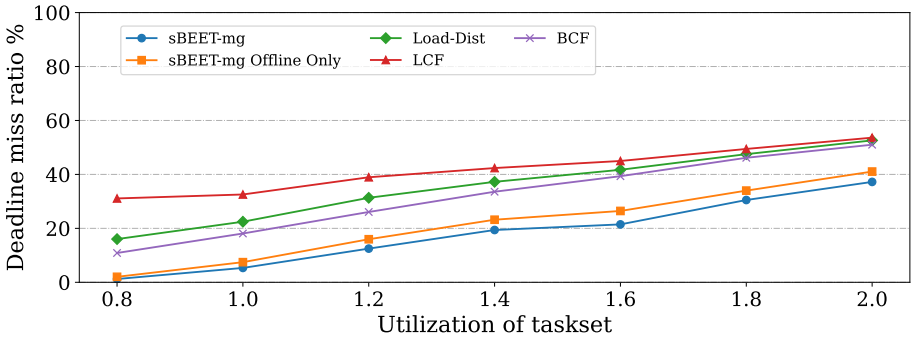
Table 3.5: Parameters for taskset generation

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

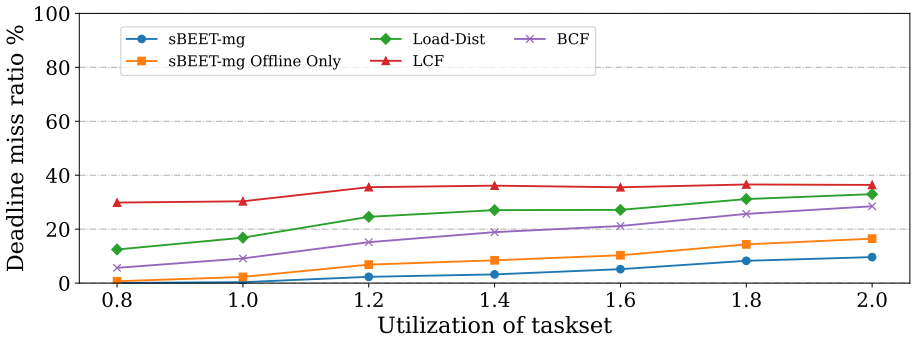
[6] We define a "big-core" as a GPU with higher computational capacity and a "little-core" as a GPU with lower capacity, i.e. more and fewer SMs.



(a) 6 SMs allowed on RTX3070



(b) 12 SMs allowed on RTX3070



(c) 24 SMs allowed on RTX3070

Figure 3.7: Deadline miss ratio w.r.t. the utilization of taskset

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 3.5 using the UUnifast algorithm [16], 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. 3.6.1, 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. 3.7 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 loaded when SMs on RTX3070 is limited to 6 as Fig. 3.7a shows, and least loaded when SMs on RTX3070 is limited to 24 as Fig. 3.7c shows. We can see that the deadline miss ratio under all methods is getting lower from the top figure to the bottom. In Fig. 3.7a, 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. 3.7b and 3.7c, 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.

Results of Energy Consumption

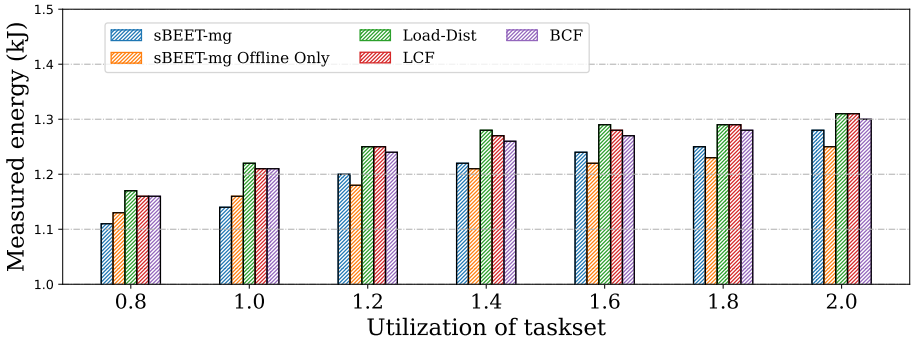
While running the experiments in Sec. 3.6.1, we also measured the runtime energy consumption of the five approaches, and the results are shown in Fig. 3.8. At first, we can observe that, with 24 SMs on RTX3070 and $U \leqslant 1.0$, BCF yields marginally better energy consumption than sBEET-mg. This is because BCF assigns all workloads to the bigger

Table 3.6: Power prediction for tasksets with different utilizations

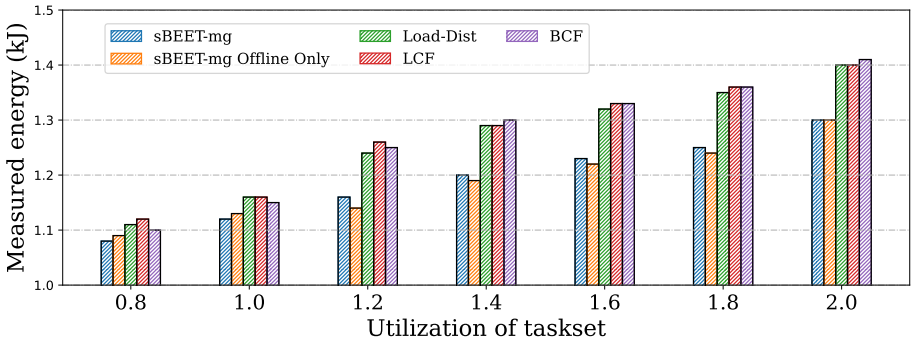
** Note: Maximum power is ≈ 180 W*

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

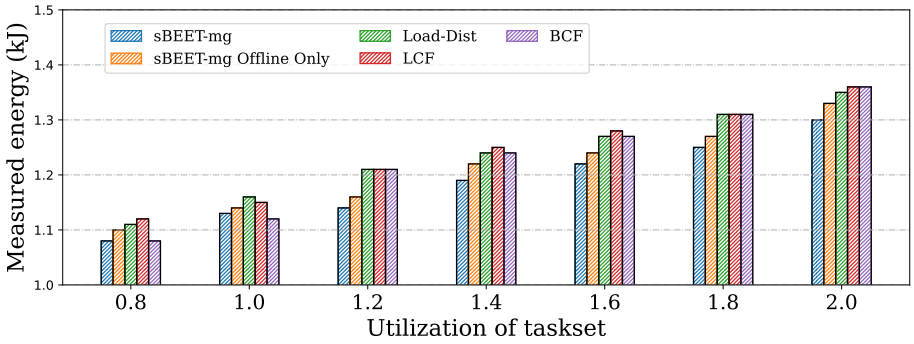
GPU (RTX3070) and leaves the smaller GPU (T400) idle all the time; however, it causes an excessively high number of deadline misses, as shown in Fig. 3.7c. In the other cases, the energy consumption of sBEET-mg and sBEET-mg Offline Only is always lower than the other three approaches that are energy-agnostic. Under all the three SM configurations with $U \leqslant 1.0$, 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 $U \geqslant 1.2$, 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 [68].



(a) 6 SMs allowed on RTX3070



(b) 12 SMs allowed on RTX3070



(c) 24 SMs allowed on RTX3070

Figure 3.8: Energy consumption w.r.t. taskset utilization

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. 3.4.1. For each utilization considered,

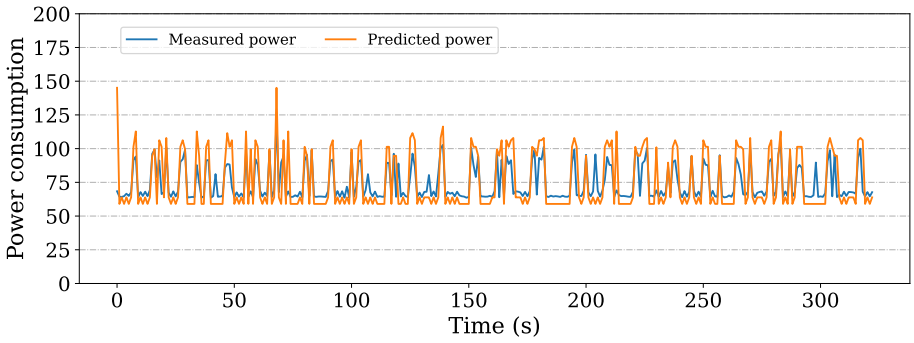


Figure 3.9: Trace of actual and predicted power consumption

we randomly select one taskset consisting of 6 tasks from the benchmark pool using the parameters given in Table 3.5, and run each taskset using our proposed scheduler for 5 minutes. Fig. 3.9 illustrates the measured and estimated power traces of a taskset with utilization of 1.0. Table. 3.6 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.

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 3.5 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. 3.10 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.

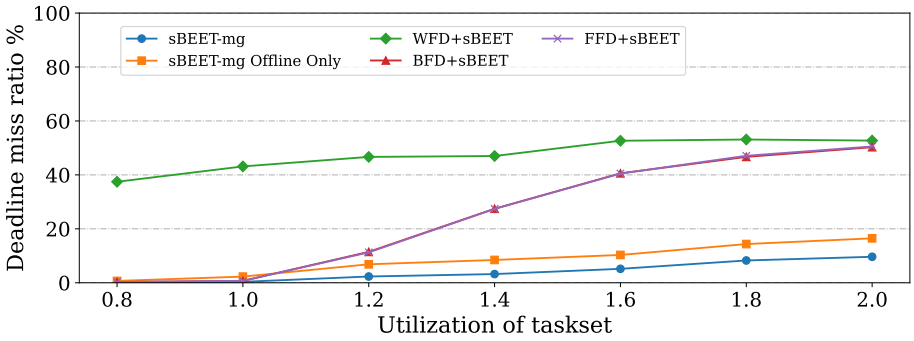


Figure 3.10: Deadline miss ratio of sBEET-mg and sBEET

Effect of Job Migration

To better understand the effect of runtime job migration, let us consider the following two case studies.

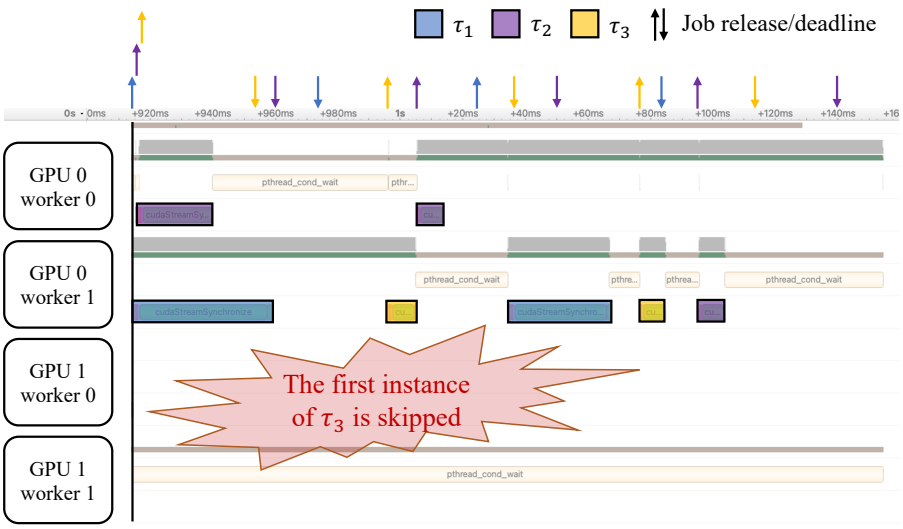
Case Study 1. Fig. 3.11 depicts the execution traces of the taskset listed in Table 3.7 under sBEET-mg with and without job migration. The trace was collected using Nvidia Nsight Compute. The task-related GPU activities are highlighted in different colors. For this taskset, all tasks are assigned to RTX3070 by Alg. 4 due to the energy efficiency consideration. However, they are not schedulable when job migration is not used; as noted

in Fig. 3.11a, $J_{3,1}$ is skipped. Fig. 3.11b shows the case where job migration is enabled. Unlike the previous case, when $J_{1,1}$ arrives, the line 4 of Alg. 5 finds that the schedule would not be feasible if $J_{1,1}$ is executed with m^{opt}. Hence, it jumps to line 17 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 28 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.

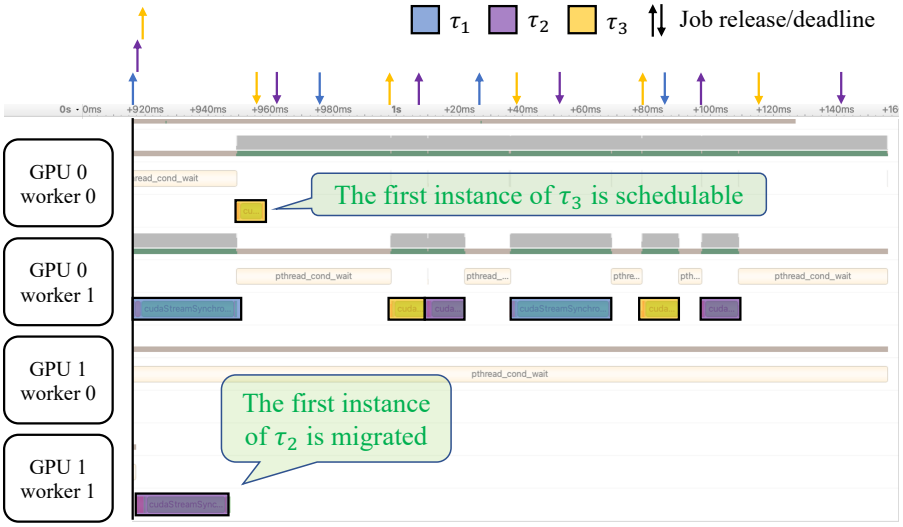
Table 3.7: Taskset used in case study 1

Task	$D_i = 0.5 * T_i$ (ms)	Offset (ms)	GPU assigned by Alg. 4
τ_1	60	0	RTX3070
τ_2	45	1	RTX3070
τ_3	40	2	RTX3070

Case Study 2. The taskset used in this case study is listed in Table 3.8 and the execution traces are shown in Fig. 3.12. For this taskset, τ_1 and τ_2 are assigned to RTX3070 and T400, respectively, by Alg. 4. In Fig. 3.12a where migration is not used, $J_{1,1}$ and $J_{2,1}$ run on their assigned GPUs exclusively. In Fig. 3.12b, when $J_{2,1}$ arrives, Alg. 5 decides to move it to another GPU to run concurrently with $J_{1,1}$ for energy efficiency (line 9 to 20. 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.



(a) sBEET-mg w/o migration



(b) sBEET-mg

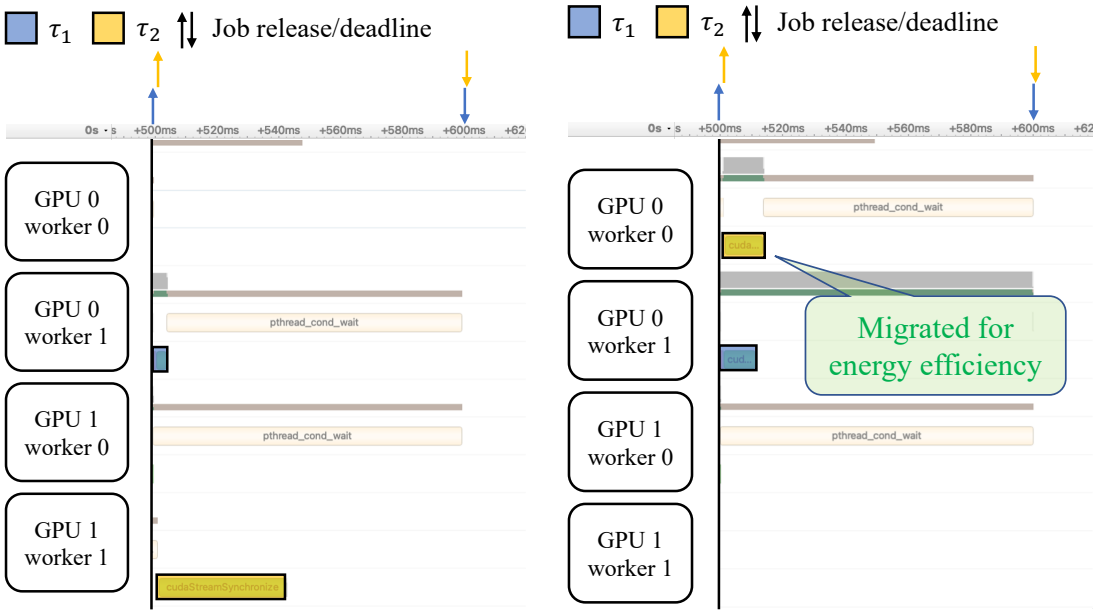
Figure 3.11: Job migration case study 1

3.6.2 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 to compare our proposed method and the baselines.

Table 3.8: Taskset used in case study 2

Task	$D_i = 0.5 * T_i$ (ms)	Offset (ms)	GPU assigned by Alg. 4
τ_1	100	0	RTX3070
τ_2	100	1	T400

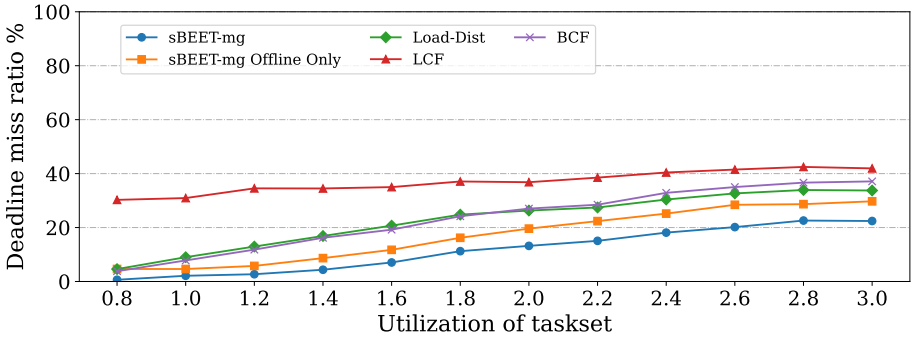


(a) sBEET-mg w/o migration

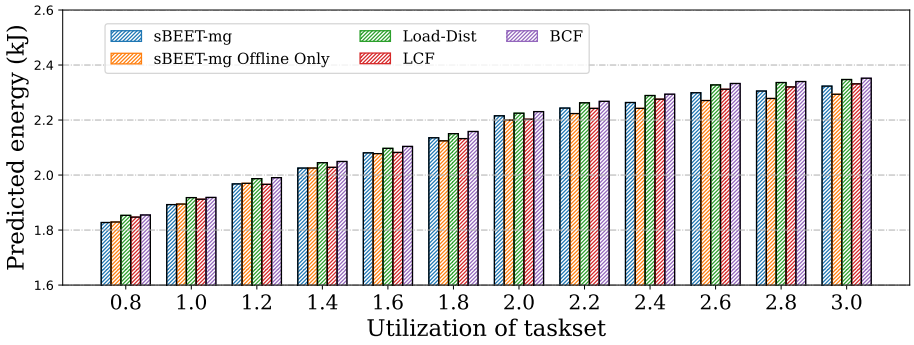
(b) sBEET-mg

Figure 3.12: Job migration case study 2

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 3.9. With parameters given in Table 3.5, 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. 3.13. The proposed method has the best schedulability among the five methods, and in most cases, sBEET-mg and sBEET-mg Offline Only have better energy consumption compared to the other baselines. The reason why sBEET has higher energy



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



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

Figure 3.13: Simulation results of GPU configuration in Table 3.9

consumption than sBEET-mg Offline Only when $U \geqslant 1.6$ is due to its better schedulability, as discussed in Sec. 3.6.1.

Table 3.9: GPU configurations in simulation

GPU Id	GPU	M_k	M_k^{limit}
π_0	RTX3070	46	12
π_1	RTX3070	46	12
π_2	T400	6	6

3.7 Conclusion

In this work, 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 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.

Chapter 4

Unleashing the Power of Preemptive Priority-Based Scheduling for Real-Time GPU Tasks

4.1 Introduction

Real-time cyber-physical systems with GPU workloads have become increasingly prevalent in various domains including self-driving cars, autonomous robots, and edge computing nodes. This trend has been accelerated in recent years by the demand for learning-enabled components as most of their implementations heavily rely on the GPU stack. The scheduling problem of GPU-using tasks in these systems is therefore crucial to ensure timely

execution and to meet stringent timing requirements. One of the key challenges here is effectively supporting prioritization and preemption, allowing higher-priority tasks to interrupt and temporarily suspend lower-priority GPU tasks whenever needed. This is particularly important in scenarios where critical high-priority tasks with stringent deadlines need to access GPU resources, while low-priority and best-effort tasks can tolerate such preemption to accommodate their execution.

As of yet, the default scheduling policy of commercial GPU devices provides little control over the prioritization and preemption of GPU tasks, causing unpredictable task response time and instability in real-time systems. The real-time research community has recognized this issue since the early era of GPU computing and has proposed several solutions. In particular, the use of real-time synchronization protocols, such as MPCP [56, 57] and FMLP+ [20], has been recognized as a promising way to manage GPU tasks in real-time systems with strong analyzable guarantees on the worst-case task response time. However, these approaches can suffer from long blocking time and priority inversion by lower-priority tasks since GPU access segments are handled non-preemptively. There have been attempts to support priority-based GPU scheduling with preemption capabilities [14, 42, 75], but they require significant modifications to GPU access code, lack analytical support, and more importantly, may not work properly if the system has processes with unmodified GPU code or graphics applications due to the time-shared GPU context switching behavior of the device driver [13, 24].

In this work, we address the aforementioned challenges and limitations by proposing novel preemptive priority-based GPU scheduling approaches for real-time GPU task

execution in multi-core systems with analyzable guarantees. Our work focuses on Nvidia GPUs, especially those on Tegra System-on-Chips (SoCs) used in embedded platforms like Jetson Xavier and Orin. We propose two distinct approaches: kernel thread and IOCTL-based approaches, each offering unique advantages with different performance implications. These approaches work at the device driver level, and unlike existing techniques, they can protect the execution of real-time GPU processes from interference from best-effort non-real-time CUDA processes and graphics processes in the system. Specifically, the kernel-thread approach requires no modifications to user-level GPU code (both host and kernel code) at all, making it amenable to use with any type of workloads. This is particularly appealing to recent machine learning and computer vision applications as they are built on top of massive libraries that involve hundreds of different kernels. The IOCTL-based approach, on the other hand, requires a small modification to GPU access code, i.e., adding just one macro at the boundaries of GPU segments, but provides more fine-grained and efficient control of the GPU. Thanks to the strictly preemptive and priority-driven GPU scheduling behavior, both approaches are analyzable and allow us to derive response-time tests for schedulabiltiy analysis.

4.2 Background on Tegra GPU Scheduling

Computational GPU workloads for Nvidia GPUs are often programmed using the CUDA library. These workloads are represented in *kernels* and user-level processes can launch kernels to the GPU at runtime. CUDA provides processes with *streams* to enable concurrent execution of kernels with a limited number of stream priority levels, e.g., only 2

in the Pascal architecture [73]. Since streams are bound to a user-level process that created them, the effect of stream scheduling and stream priority assignment is exerted only within each process boundary. The CUDA library is not a must for processes to access the GPU hardware. There are other low-level libraries for general-purpose GPU computing and graphics applications such as OpenCL and Vulkan. Programs built using different libraries co-exist in the system and they send GPU commands to the device driver.

At the device driver level, each process is assigned a unique *GPU context*, which encompasses a virtual address space and other runtime states on the GPU side. Regardless of whether a process utilizes the CUDA library in the user space, it will have its own dedicated GPU context. The GPU contexts from different processes are time-sliced to share the GPU hardware. To ensure fairness and prevent resource contention, the Tegra GPU driver uses a scheduling policy that assigns entries in the "runlist".[1] The entries of the runlist represent the allocation of time slices to TSGs (Time-Sliced GPUs) that are directly associated with processes. Fig. 4.1 illustrates the runlist filled with TSG entries. The TSG data structure maintains various attributes, including the process ID (pid), a list of channels, and the duration of the time slice. Each channel contains a stream of GPU commands received from its process. The scheduling of the runlist follows a round-robin approach, ensuring that during each time slice, the GPU executes commands pertaining to the corresponding GPU context. The number of entries assigned to a TSG on the runlist is contingent upon its priority. However, at the time of writing, no user-space interface is provided to configure the length of the time slice or the priority settings for TSGs.

[1]In fact, there are multiple runlists but we refer to them as singular for simplicity. When scheduling, the driver cycles through each runlist to handle a higher volume of workloads, and this does not affect our proposed design.

The construction of the runlist in the Tegra GPU driver follows multiple steps. First, processes submit their commands to specific channels associated with their TSGs. Once the commands are submitted, the corresponding TSGs are added to the runlist which is protected by a mutex lock. During the construction of the runlist, TSGs with higher priority are granted a larger time slice and more entries on the runlist. After construction, the runlist is repeatedly scheduled by the GPU in a round-robin manner. Each entry on the runlist runs for its time slice, and once timeout, the TSG of the next entry is executed. This repeats until all the commands of all active TSGs on the runlist are consumed.

In summary, the Tegra GPU driver employs a *time-sliced round-robin* scheduling approach. This approach, however, does not respect the OS-level scheduling priority of processes, which is the main control knob to tune real-time performance in practice. This causes high-priority real-time tasks to experience unpredictable waiting time when the system accepts new best-effort tasks. In addition, it is not easy for the user to observe such driver-level behavior because GPU profiling tools, such as Nvidia Nsight Systems, do not report GPU context switching events and each kernel execution time appears to be inflated with no time slice information. These issues contribute to difficulties in understanding and predicting the runtime behavior of GPU-enabled real-time systems.

4.3 Related Work

Table 4.1 gives a summary of comparison between representative GPU scheduling approaches. Below we discuss prior work in various categories.

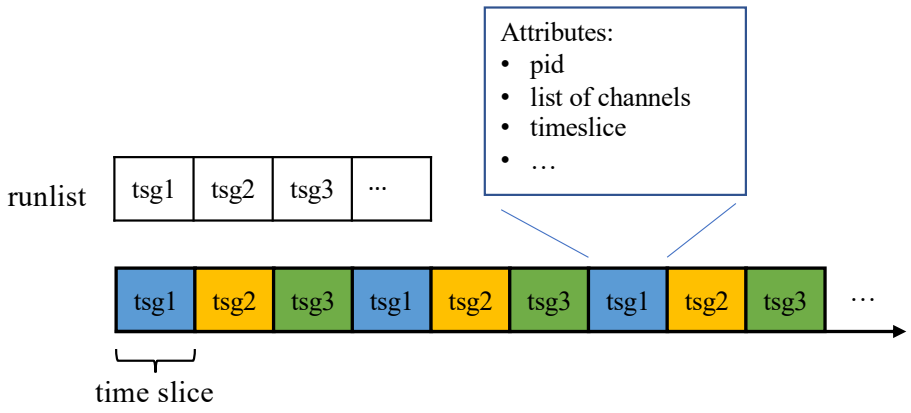


Figure 4.1: Runlist and time-sliced GPU scheduling

		No blocking	Task priority respected	No source code modification	Analyzable
Prior work	Unmanaged GPU (default driver)	×	×	✓	?
	Sync.-based approaches	×	✓	×	✓
Ours	Kernel thread approach	✓	✓	✓	✓
	IOCTL-based approach	✓	✓	×	✓

Table 4.1: Comparison of different GPU scheduling approaches

Synchronization-Based GPU Access Control. Real-time synchronization protocols have played an important role in managing access to GPUs [28–30, 56]. With this approach, GPUs are modeled as mutually-exclusive shared resources and tasks are made to acquire locks to enter code segments accessing the GPUs, i.e., critical sections. MPCP [57] and FMLP+ [19] are prime examples for multi-core systems with GPUs and the use of such protocols enables analytically provable worst-case task response time bounds. However, the synchronization-based approach may suffer from blocking time from lower-priority tasks holding a lock and priority inversion caused by the priority boosting mechanism employed

in these protocols [44]. This becomes particularly problematic when tasks busy-wait on long kernel execution, as discussed in [56].

Preemptive GPU Scheduling. Several previous studies [14, 42, 75] have proposed software-based mechanisms to enable preemptive scheduling of real-time GPU tasks. These approaches introduce the concept of decomposing long-running GPU kernels into smaller blocks, allowing preemption to occur at the boundaries of these blocks. By enabling preemptive scheduling, the waiting time of high-priority tasks can be significantly reduced, improving responsiveness and offering a better chance to meet timing requirements. However, the cost of utilizing these mechanisms is not trivial as they necessitate a significant rewriting of user programs [14] or an implementation of a custom CUDA library with device driver modifications [14, 75]. Capodieci et al. [24] proposed a hypervisor-based technique to support preemptive Earliest Deadline First (EDF) GPU scheduling of virtual machines (VMs) in a virtualized environment. This approach achieves GPU performance isolation among VMs and shares some similarities with our work, in terms of controlling GPU context switching at the device driver level. However, it lacks consideration of the end-to-end response time of tasks involving CPU and GPU interactions, which is a specific focus of our work. Recently, Han et al. [31] proposed REEF, which enables microsecond-scale, reset-based preemption for concurrent DNN inferences on GPUs. This approach proactively kills and restarts best-effort kernels leveraging the idempotent nature of most DNN inference, but it is not applicable to a wide range of applications.

GPU Partitioning. As a GPU is composed of multiple compute units, e.g., Streaming Multiprocessors (SMs) on Nvidia GPUs, there has been attempts to spatially partitioning

the GPU and making them accessible by multiple real-time tasks in parallel [38, 58, 67, 68, 76]. They use SM-centric kernels transformation [72] to run kernels on their designated SMs/partitions. As this involves extensive program modifications and may suffer from misbehaving tasks, Bakita and Anderson [13] recently proposed a user-space library that minimizes program changes and offers much better usability and portability. However, all these approaches work within a single GPU context, i.e., one process. Hence, multiple processes with separate contexts will still time-share the GPU, as discussed in Sec. 4.2. Our work overcomes this issue and can be co-used with these partitioning techniques.

4.4 System Model

We consider a multi-core system with a GPU, which is common in today's embedded hardware platforms like Nvidia Jetson. The CPU has ω identical cores and the GPU is yet another processing resource used by compute-intensive tasks. The GPU consists of internal resources including Execution Engines (EEs) and Copy Engines (CEs). The EE and CE operations of a single process can be done asynchronously at runtime, and during pure GPU execution, the process can either busy-wait or self-suspend on the CPU. However, different processes cannot use the GPU at the same time because of the time-sharing scheduling of GPU contexts at the GPU device driver, as discussed before.

Task Model. We consider a taskset Γ consisting of n periodic tasks (processes) with fixed priority and constrained deadlines.[2] Each task is assumed to be preallocated to one CPU core with no runtime migration, i.e., partitioned multiprocessor scheduling. The execution

[2]We assume tasks are processes and use them interchangeably in this paper.

of a task is an alternating sequence of CPU segments and GPU segments. CPU segments run entirely on the CPU and GPU segments involve GPU operations such as memory copy and kernel execution. A task τ_i can be characterized as follow:

$$\tau_i := (C_i, G_i, T_i, D_i, \eta_i^g)$$

- C_i: the cumulative sum of the worst-case execution time (WCET) of all CPU segments of task τ_i.
- G_i: the cumulative WCET of GPU segments (including memory copies and kernels) of τ_i.
- T_i: the minimum inter-arrival time of each job of τ_i.
- D_i: the relative deadline of each job of τ_i, and is smaller than or equal to the period, i.e., $D_i \leqslant T_i$.
- η_i^c: the number of CPU segments in each job of task τ_i.
- η_i^g: the number of GPU segments in each job of task τ_i; if τ_i does not use the GPU, $\eta_i^g = 0$.

Fig. 4.2 depicts these parameters. We use $G_{i,j}$ to denote the WCET of the j-th GPU segment of task τ_i, i.e., $G_i = \sum_{j=1}^{\eta_i^g} G_{i,j}$. Each GPU segment $G_{i,j}$ includes memory copy and kernel execution, and can be characterized as follow:

$$G_{i,j} := (G_{i,j}^m, G_{i,j}^e)$$

- $G_{i,j}^m$: the WCET of miscellaneous operations in the j-th GPU segment of task τ_i that require CPU intervention.

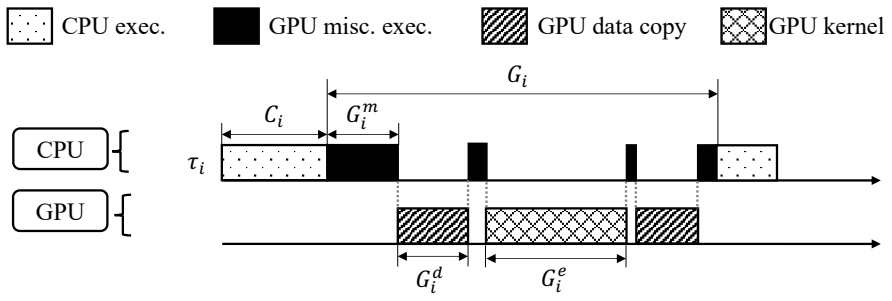


Figure 4.2: Task model example

- $G^e_{i,j}$: the WCET of GPU operations in the j-th GPU segment that requires *no* CPU intervention, and we call it a *pure GPU segment*.

$G^m_{i,j}$ is the time for launching a CUDA kernel, overhead for communicating with the GPU driver, and miscellaneous CPU operations for issuing other GPU commands. $G^e_{i,j}$ is the time for GPU data copy and kernel execution, during which task τ_i can either busy-wait or self-suspend on the CPU. Note that $G_{i,j} \leqslant G^m_{i,j} + G^e_{i,j}$ because the worst-case of G^m and G^e are not necessarily happening on the same control path and they may execute in parallel in asynchronous mode [56].

We also consider best-case execution time, denoted by a check mark (e.g., $\breve{C}_i$), to improve our analysis in Sec. 4.6.2. For readability, we will explain the parameters that follow this notation where they are used.

4.5 Priority-Based Preemptive GPU Scheduling

We present two runtime approaches, kernel thread and IOCTL-based, for preemptive priority-based execution of GPU segments from real-time tasks. The first approach involves a kernel thread that polls for any changes in the status of tasks and updates the

Algorithm 6 Kernel Thread Approach

```
1: procedure KERNELTHREADRUNLISTUPDATE
2:     while true do
3:         if τ_i's state is changed from the previous cycle then
4:             τ_h ← the highest-priority GPU-using ready task
5:             if τ_h exists then
6:                 Add τ_h's associated TSGs to runlists
7:                 Remove other TSGs from runlists
8:             else                                  ▷ no RUNNING real-time tasks
9:                 Add all active TSGs to runlists
10:        Wait for the next polling cycle
```

runlist accordingly. The second approach involves a set of user-level runtime macros that notify the GPU driver to update the runlist.

The kernel thread approach is easier to use as it does not require any modification to program code, but it makes scheduling decisions and updates the runlist only at job execution level, and this may lead to resource underutilization. The IOCTL-based approach provides finer-grained control over GPU segments, but this requires user-level code modification although small. The details of these two approaches are presented in this section.

4.5.1 Kernel Thread Approach

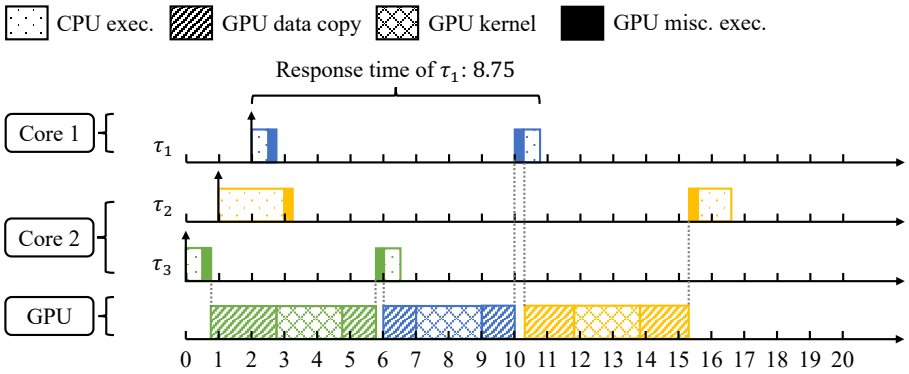
The kernel thread approach creates a kernel thread that is initiated along with the driver software. It continuously polls for changes in task status (`task_struct::state`),

with a predefined time interval of 1 millisecond on a designated CPU core, to avoid excessive CPU usage and minimize interference to other tasks. When a task status change is detected, e.g., from `TASK_RUNNING` to `TASK_STOPPED`, it updates the runlist.

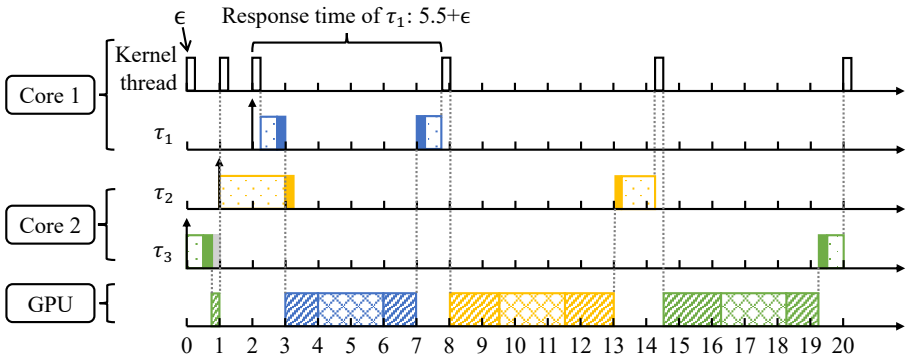
The procedure is shown in Alg. 6. At every polling cycle, the kernel thread checks if any task τ_i with an active TSG has changed its state from the previous cycle (line 3). If yes, it obtains a ready task (`TASK_RUNNING`) with the highest real-time priority (`task_struct::rt_priority`) as τ_h (line 4). Next, the scheduler decides whether the runlist should be updated. If τ_h exists, the scheduler removes all other TSGs from the runlist but only keeps τ_h's associated TSGs in it. Otherwise, it means that no GPU-using real-time task is ready to run, and in this case, the scheduler puts all other active TSGs back into the runlist, allowing non-real-time best-effort tasks to make their progress (lines 5 to 9).

However, since the scheduling decision is made only when a task state changes, this approach may underutilize GPU resources. For instance, when a high-priority task τ_h starts to run, the currently-running task τ_i's TSGs are removed from the runlist to reserve the GPU for τ_h. However, while τ_h is executing its CPU segments, the GPU may remain idle, leading to underutilization of GPU resources. The kernel thread approach has another limitation. Self-suspension during GPU execution is not allowed and a task must spin on the CPU side to maintain its task state. This is because task state changes due to self-suspension can be misinterpreted by the kernel thread, and may make incorrect scheduling decisions and cause unnecessary runlist updates.

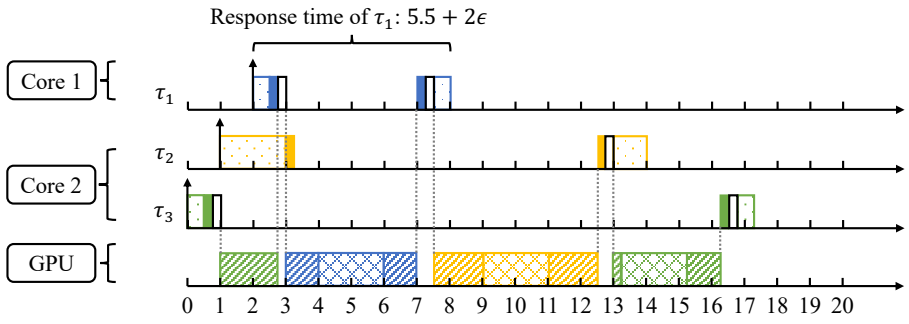
Figs. 4.3a and 4.3b compare task schedules under the conventional synchronization-based approach and our kernel thread approach. τ_1 is running on Core 1, while τ_2 and τ_3



(a) Schedule under synchronization-based approach



(b) Schedule under kernel thread approach



(c) Schedule under IOCTL-based approach

Figure 4.3: Example schedule of three tasks under different approaches (priority $\tau_1 > \tau_2 > \tau_3$)

are running on Core 2. The synchronization-based approach shown in Fig. 4.3a treats the entire execution of a GPU segment as a critical section. Tasks are serviced in order based on their task priorities. This approach ensures that each task completes its GPU segments in a deterministic and predictable manner. However, as can be seen in the figure, τ_1 is delayed by the GPU segments of all of its lower-priority tasks and gets a response time of 8.75. On the other hand, our kernel thread approach avoids this delay by allowing preemption during GPU segment execution. In Fig. 4.3b, the kernel thread is on Core 1 along with τ_1 and the runlist update time is denoted as ϵ.[3] At $t = 1$, the kernel thread updates the runlist and causes preemption of τ_3's GPU segment by removing its associated TSGs from the runlist. Task τ_1 is delayed by ϵ due to running on the same core as the kernel thread. The GPU is then allocated to the highest-priority task, τ_1, until it completes. The response time of τ_1 is $5.5+\epsilon$, much smaller than that of the synchronization-based approach.

Although it is not depicted in the above figure, there could be a delay for GPU preemption to take effect because Nvidia GPUs support preemption at the pixel level for graphics tasks and the thread-block level for compute tasks [11]. Such delay is thus very small compared to the length of GPU kernels, and for compute tasks, it can be separately measured or estimated by the maximum length of a single thread block among all kernels. We assume that ϵ includes this delay in it.

[3]Prior work [24] reports that the runlist update overhead including GPU context switching can take from 50 to 750 μs. Our analysis in Sec. 4.6 takes into account ϵ and our measurements in Sec. 4.7.2 show similar results.

4.5.2 IOCTL-Based Approach

The IOCTL-based approach is a user-level runtime method for efficient control of GPU segments in the runlist. To implement this method, we add two macros that allow user programs to indicate the beginning and completion of a GPU segment. When the macro is called, it generates an IOCTL command and sends it to the GPU driver through a file descriptor, and requests the driver to update the runlist accordingly.

```
1  int main() {
2      ...
3      cudaStream_t stream;
4      ...
5      cudaStreamBegin(getpid());
6      cudaMemcpyAsync(d_in, h_in, mem_sz_in, cudaMemcpyHostToDevice, stream);
7      MyKernel<<<grid, threads, 0, stream>>>(d_in, d_out, dims_in, dims_out);
8      cudaStreamEnd(getpid(), stream);
9      ...
10 }
```

Listing 4.1: Example Usage of IOCTL-based approach

The macros introduced are `cudaStreamBegin()` and `cudaStreamEnd()`, which are wrappers to our IOCTL syscalls. A sample user program is listed in Listing 4.1. The code between them is a GPU segment. Unlike the kernel thread approach in which the runlist update is triggered at the boundaries of each task's execution, with the help of these two macros, we can define the boundaries of GPU segments and allows GPU segments and CPU segments from different tasks to be co-scheduled. In the Tegra driver, the default runlist update is protected by a mutex lock. As the IOCTL-based approach allows multiple tasks

to make these calls concurrently, We replace the default one with a priority-based lock to reduce the blocking time.

The procedure to update the runlists under this approach is shown in Alg. 7. To track which tasks are in the runlist and which tasks are pending, two bitmaps are maintained in the GPU driver. The procedure is started by an IOCTL command which is wrapped by our macro. When a caller task τ_i requests to be added to the runlist (through `cudaStreamBegin()`), the scheduler implemented inside the IOCTL function first checks whether τ_i is a real-time task. If it is not, the scheduler checks for any currently running real-time tasks and decides whether to add τ_i to the runlists or to the pending list (line 5 to 9). If τ_i is a real-time task, the scheduler compares the priority of τ_i with the currently running task τ_h. If τ_i has a higher priority, the scheduler preempts the GPU execution of τ_h, moving it to the pending list, and adds τ_i to the runlist. Otherwise, τ_i is added to the pending list (line 10 to 16). Upon completion of τ_i signaled by `cudaStreamEnd()`, the scheduler examines the pending list for the highest-priority task τ_k. If τ_k exists, it is added to the runlist. Otherwise, if only best-effort tasks remain, they are added to the runlist to resume their progress (line 17 to 24).

Fig. 4.3c shows an example schedule under the IOCTL-based approach using the same taskset as in the previous section. Unlike the kernel thread approach, τ_3's GPU segments are not preempted until τ_1 starts its GPU kernel execution. This strategy is followed in the remaining schedule. In this case, the response time of task τ_1 is $5.5 + 2\epsilon$.

The IOCTL-based approach allows fine-grained GPU resource control and ensures prompt execution of high-priority tasks. However, implementing this approach necessitates

Algorithm 7 IOCTL-based Approach

```
1: task_pending = ∅
2: task_running = ∅                    ▷ Note that a task exclusively exists in these two lists
3: procedure IOCTLRunlistUpdate(τ_i, add)
4:     if add then                                                  ▷ τ_i requests to be added
5:         if τ_i is not a real-time task then
6:             if no real-time task is in task_running then
7:                 Add τ_i to task_running
8:             else
9:                 Add τ_i to task_pending
10:        else                                                     ▷ τ_i is a real-time task
11:            τ_h ← the highest-priority task in task_running
12:            if τ_i → prior > τ_h → prio then
13:                Add τ_i to task_running
14:                Add τ_h to task_pending
15:            else
16:                Add τ_i to task_pending
17:    else                                                         ▷ τ_i requests to be removed
18:        τ_k ← the highest-priority task in task_pending
19:        if τ_k exists then
20:            Add τ_k to task_running
21:            Remove τ_i from task_running
22:        else                                                     ▷ no pending real-time task
23:            task_running ← task_pending
24:            task_pending ← ∅
25:    Add all TSGs associated with tasks in task_running to runlists
```

modifying user-level code, which could pose a challenge in terms of adoption and practical implementation.

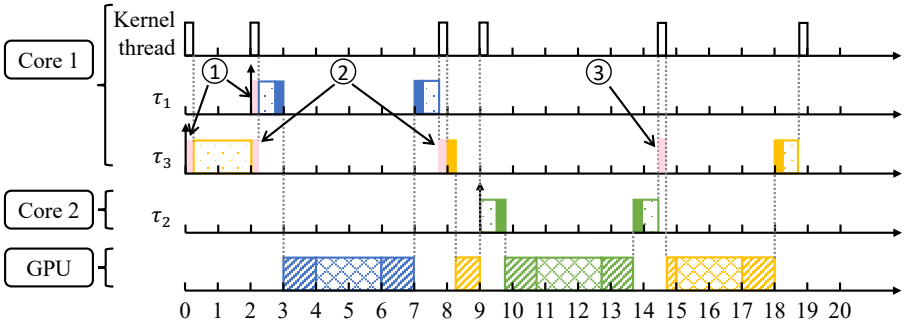
4.5.3 GPU Segment Priority Assignment

In both kernel thread and IOCTL-based approaches, GPU segments are executed following their OS-level task priorities. In the kernel thread approach, the kernel thread has the highest priority, and the preemption occurs at job execution boundaries. In the IOCTL-based approach, preemption can occur at segment boundaries.

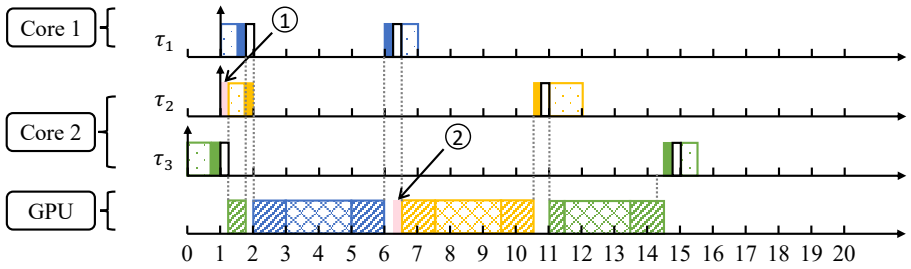
To improve taskset schedulability, we can assign separate priority to the GPU segments of a task, different from its CPU priority. We adopt Audsley's approach for this purpose [12]. Hence, if the schedulability test given in the next section determines a taskset is unschedulable, we iterate through all tasks from the lowest to the highest CPU priority and check whether each priority level can be assigned to the GPU segments of a task without causing the taskset to fail the schedulability test.

4.6 Schedulability Analysis

In this section, we present a comprehensive analysis of schedulability for preemptive GPU scheduling. We first give analysis under two approaches mentioned in the previous section. We then introduce a technique to reduce the pessimism of our schedulability analysis and provide a tighter bound on the worst-case response time.



(a) Kernel thread approach



(b) IOCTL-based approach

Figure 4.4: Example schedule of three tasks with runlist update delay (task priority: $\tau_1 > \tau_2 > \tau_3$)

4.6.1 Baseline Analysis

Our baseline analysis provides a conservative upper bound on the worst-case response time under the two proposed approaches. In our model, preemptions can occur in two scenarios: (i) *CPU preemption*: a CPU segment of a task τ_i is preempted by a CPU segment of a higher-priority task τ_h running on the same CPU core, and (ii) *GPU preemption*: a GPU segment of a task τ_i is preempted by a GPU segment of a higher-priority task τ_h, regardless of which CPU core is assigned to τ_h. Based on these scenarios, we develop a schedulability analysis for the two proposed approaches in the next subsection.

Preemptive GPU Under Kernel Thread Approach

The kernel thread runs on one designated CPU core and updates the runlist on behalf of other GPU-using tasks. Task self-suspension is not applicable here, and only the busy-waiting mode is available to monitor task states and prevent incorrect runlist updates in the middle of job execution.

We first identify the delay for a task τ_i caused by the runlist updates of the kernel thread. The following lemma holds for this delay:

Lemma 12 *The runlist update delay from the kernel thread for a job of task τ_i is upper-bounded by:*

$$K_i = x_i \cdot (2\epsilon + \sum_{\tau_h \in hp(\tau_i)} \lceil \frac{R_i + J_h}{T_h} \rceil \cdot 2\epsilon) \tag{4.1}$$

where

$$x_i = \begin{cases} 1 & \text{, } \tau_i \text{ is a GPU-using task } (\eta_i^g > 0) \text{ and runs on the} \\ & \text{ same core as the kernel thread} \\ 0 & \text{, otherwise} \end{cases}$$

, ϵ is the runlist update time (Sec. 4.5.1), R_i is the worst-case response time of τ_i, $hp(\tau_i)$ is a set of all the higher-priority tasks than τ_i in the system, and $J_h = R_h - (C_h + G_h)$ is the release jitter to capture the carry-in effect.

Proof. Whenever the kernel thread updates the runlist, it delays the CPU execution of other tasks on the same core due to its highest priority and also the GPU execution due to TSG evictions and GPU context switching [24]. Hence, GPU-using tasks on any CPU core and CPU-only tasks running on the same core as the kernel thread are subject

to this delay. The only exception is CPU-only tasks running on a different core as they are neither delayed by the CPU and GPU operations of the kernel thread (the x_i term).

Once the job of τ_i starts execution, its status change triggers the kernel thread once to update the runlist. The kernel thread might be already updating the runlist for a lower-priority task, so one additional ϵ needs to be considered (the first term in the parenthesis). During τ_i's job execution (R_i), additional invocation of the kernel thread is determined by only higher-priority jobs since those with lower priority than τ_i cannot trigger the runlist update until the completion of the τ_i's job. The number of arrivals of high-priority tasks during R_i is upper-bounded by $\lceil \frac{R_i+J_h}{T_h} \rceil$, where adding J_h is a known method to capture a carry-in job in an arbitrary interval [15]. Each high-priority job involves two times of runlist updates (2ϵ), one at the beginning of the high-priority job and another at its completion to resume τ_i's job. ■

Fig. 4.4a gives an example of all types of delay caused by the kernel thread. Tasks τ_1, τ_3, and the kernel thread are running on Core 1, and τ_2 is running on Core 2. The taskset is with the priority of $\tau_1 > \tau_2 > \tau_3$. ① is the delay that occurs when any task running on the same core as the kernel thread has a state change, i.e., starting job execution. During the preemption of τ_1 on τ_3, ② illustrates the delay caused by runlist updates before and after τ_1 completes. Lastly, ③ demostrates the delay by a remote preemption from τ_2 on τ_3.

Lemma 13 *Under the kernel thread approach, the worst-case response time of a task τ_i is upper-bounded by:*

$$
\begin{aligned}
R_i = C_i + G_i + K_i \\
&+ \sum_{\tau_h \in hpp(\tau_i)} \lceil \frac{R_i}{T_h} \rceil (C_h + G_h) \\
&+ \sum_{\substack{\eta_i^g > 0 \wedge \eta_h^g > 0 \\ \wedge \tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i)}} \lceil \frac{R_i + J_h}{T_h} \rceil (C_h + G_h)
\end{aligned}
\tag{4.2}
$$

where $hpp(\tau_i)$ is the set of higher-priority tasks running on the same CPU core as τ_i.

Proof. This is an extension of the conventional iterative response-time test. C_i and G_i denote the execution time of the CPU segments and the GPU segments of τ_i, respectively. K_i bounds the delay from the kernel thread. By the design of the kernel thread approach, higher-priority tasks τ_h on the same CPU core as τ_h can preempt τ_i for their entire job execution $(C_h + G_h)$ because τ_i busy-waits on the CPU (the second line of the equation). Higher-priority GPU-using tasks ($\eta_h^g > 0$) running on different cores can also effectively preempt τ_i but only if τ_i has a GPU segment ($\eta_i^g > 0$). Carry-in jobs of higher-priority tasks on different cores are captured by J_h in the last term, as in Lemma 12.

■

Preemptive GPU Under IOCTL-Based Approach

Unlike the kernel thread approach, runlist update is done by each task by holding a lock in the GPU driver. The duration of each lock-holding time is the same as a single runlist update time, ϵ, but the use of the lock introduces a slightly different delay from K_i.

Lemma 14 *The runlist update delay under the IOCTL-based approach for a job of task τ_i is upper-bounded by:*

$$B_i = 2\epsilon \cdot \eta_i^g + \sum_{\substack{\eta_i^g > 0 \wedge \eta_h^g > 0 \\ \wedge \tau_h \in hp(\tau_i)}} \lceil \frac{R_i + J_h}{T_h} \rceil \cdot 2\epsilon \tag{4.3}$$

Proof. Recall that our lock implements a priority-based waiting list, but real-time multiprocessor synchronization such as MPCP [56] and FMLP+ [20], there is no priority boosting. Hence, as in Lemma 12, CPU-only tasks are unaffected by the runlist update. For GPU-using tasks, each GPU segment needs 2ϵ, one for the runlist update of itself and another for at most one blocking from lower-priority tasks. During R_i, the total number of runlist updates due to higher-priority tasks can be bounded in the same manner as Lemma 12. ■

Fig. 4.4b illustrates an example of the runlist update delay under the IOCTL-based approach. Tasks τ_2 and τ_3 are running on Core 1 and τ_1 is running on Core 2. The three tasks have priorities of $\tau_1 > \tau_2 > \tau_3$. We examine the blocking time that τ_2 experiences. The runlist update by τ_3 at $t = 1$ causes a blocking at ① for τ_2. Then τ_2 is preempted by the GPU execution of τ_1, and this adds an extra cost at ②. After the GPU segments of τ_i are ready to run, they get fully preempted by τ_1, as depicted by ②.

The IOCTL-based approach provides tasks with both self-suspension and busy-waiting options during pure GPU execution. We analyze the resposne time of each case below.

Lemma 15 *Under the IOCTL-based approach, the worst-case response time of a self-suspending task τ_i is bounded by:*

$$
\begin{aligned}
R_i = C_i + G_i + B_i \\
&+ \sum_{\tau_h \in hpp(\tau_i)} \lceil \frac{R_i + J_h^c}{T_h} \rceil (C_h + G_h^m) \\
&+ \sum_{\substack{\eta_i^g > 0 \wedge \eta_h^g > 0 \\ \wedge \tau_h \in hp(\tau_i)}} \lceil \frac{R_i + J_h^g}{T_h} \rceil G_h^e
\end{aligned}
\tag{4.4}
$$

where $J_h^c = R_h - (C_h + G_h^m)$ *and* $J_h^g = R_h - G_h^e$.

Proof. This is a variant of the kernel-thread analysis given by Lemma 13. The biggest difference is that, with self-suspension, higher-priority tasks on the same CPU core do not impose CPU interference during their pure GPU execution; hence, each job of τ_h gives up to $C_h + G_h^m$. The self-suspending behavior of an interfering job is known to be bounded by a release jitter given by its response time deducted by the worst-case execution time [18]. In our case, J_h^c computes that. On the GPU, the pure GPU segments (G_h^e) of all higher-priority tasks in the system need to be considered, along with the effect of carry-in jobs using J_h^g. ■

Lemma 16 *Under the IOCTL-based approach, the worst-case response time of a busy-waiting task* τ_i *is bounded by:*

$$
\begin{aligned}
R_i = C_i + G_i + B_i \\
&+ \sum_{\tau_h \in hpp(\tau_i)} \lceil \frac{R_i}{T_h} \rceil (C_h + G_h) \\
&+ \sum_{\substack{\eta_i^g > 0 \wedge \eta_h^g > 0 \\ \wedge \tau_h \in hp(\tau_i) \wedge \tau_h \notin hpp(\tau_i)}} \lceil \frac{R_i + J_h^g}{T_h} \rceil G_h^e
\end{aligned}
\tag{4.5}
$$

Proof. The proof directly follows Lemmas 13 and 15. ■

4.6.2 Analysis With Reduced Pessimism

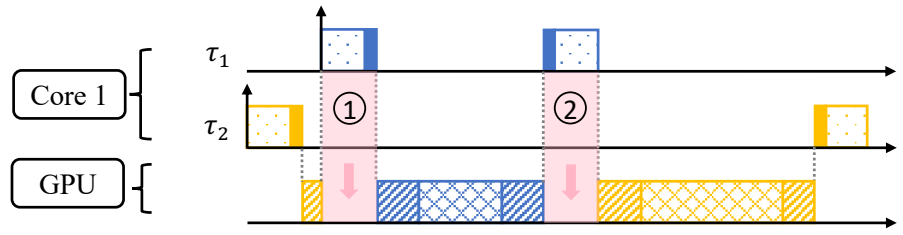
In the above analysis, the total preemption time on task τ_i caused by higher-priority tasks running on the same core is simply computed by adding up the worst-case preemption time on both CPU and GPU, assuming both types of preemptions occur throughout execution. However, there are two key factors that make this analysis more conservative than necessary:

- Both CPU and GPU preemptions are assumed to happen at their full extent.
- Under self-suspension mode, a task τ_i cannot experience full preemption of all CPU and GPU segments of a higher-priority task τ_h running on the same core.

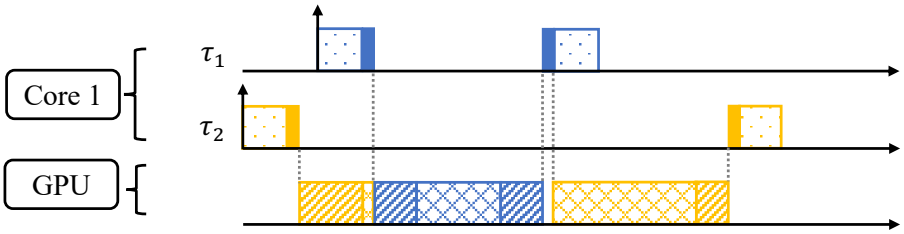
The first factor can be easily observed in Eqs. 4.2, 4.4 and 4.5 as the interval of interest of R_i is always considered when computing the number of local and remote preemptions. Such pessimism is illustrated in Fig. 4.5a. For ① and ②, the GPU execution of τ_2 should have a chance to execute together with CPU segments of τ_1, but in the baseline analysis, τ_1's CPU segments are assumed to preempt τ_2's GPU execution. Fig. 4.5b shows the case of an actual schedule where both preemptions of ① and ② do not exist.

For the second factor, for convenience, let us consider two tasks τ_h and τ_l released at the same time on the same CPU core. τ_h finishes its CPU segment first and starts GPU execution. At this time, τ_l can begin its CPU segment and there is an inevitable overlap between τ_h's GPU segment and τ_l's CPU segment, making full preemption impossible.

The baseline analysis given in Sec. 4.6.1 overestimates the worst-case response time, especially for the IOCTL-based approach that allows concurrent execution of GPU



(a) Assumed pessimistic schedule by baseline analysis. Unreal preemptions are labelled as ① and ②.



(b) Actual schedule with overlapped execution

Figure 4.5: Pessimism of baseline analysis and two types of execution overlap (IOCTL-based approach; priority: $\tau_1 > \tau_2$)

kernels and CPU segments from different tasks. We focus on reducing the pessimism by identifying minimum possible overlaps between segments, during which certain preemptions should not occur, to shrink the worst-case response time in the recursive analysis form. We establish key definitions used throughout the discussion.

Definition 17 (Completion time) *The completion time, X, of a segment is defined as the time interval between the start of execution of the segment and the completion of the segment.*

Definition 18 (Full overlap) *A group of execution segments e_1 is said to have a full overlap with a segment e_2 if all segments of e_1 are entirely contained within e_2. This means*

that the start time s_1 of the first segment of e_1 is after or equal to the start time s_2 of e_2 ($s_1 \geqslant s_2$), and the completion time c_1 of the last segment of e_1 is before or equal to the completion time c_2 of e_2 ($c_1 \leqslant c_2$). The notation $e_1 \sqsubset e_2$ denotes that e_1 fully overlaps with e_2.

Based on these, we consider two cases for task τ_i under analysis: (i) all CPU segments of a higher-priority task τ_h fully overlap with the j-th pure GPU segment of τ_i, i.e., $C_h \sqsubset G^e_{i,j}$, and (ii) all pure GPU segments of τ_h fully overlap with the j-th CPU segment of τ_i, i.e., $G^e_h \sqsubset C_{i,j}$. If the system uses Rate Monotonic or Deadline Monotonic for priority assignment, lower-priority tasks would tend to have longer periods/deadlines and execution time than those of higher-priority tasks, and finding out the minimum overlapped interval for these cases can yield nontrivial improvements.

Lemma 19 *The minimum fully overlapped CPU execution of τ_h with the j-th pure GPU segment of τ_i, i.e., $C_h \sqsubset G^e_{i,j}$, is lower-bounded by:*

$$O^{cg}_{(i,j),h} = \max((\lfloor \frac{BX^g_{i,j}}{T_h} \rfloor - 1) \cdot \breve{C}_h, 0) \tag{4.6}$$

where $BX^{cg}_{i,j}$ is the best-case relative completion time of τ_i's j-th pure GPU segment and $\breve{C}_h$ is the best-case execution time of all CPU segments of τ_h. $BX^{cg}_{i,j}$ is given by:

$$BX^g_{i,j} = \breve{G^e_{i,j}} + \sum_{\tau_h \in hp(\tau_i)} (\lceil \frac{BX^g_{i,j}}{T_h} \rceil - 1) \cdot \breve{G^e_h} \tag{4.7}$$

where the initial condition for recurrence is $BX^g_{i,j} = G^e_{i,j}$, and $\breve{G^e_{i,j}}$ is the best-case execution time of the j-th pure GPU segment of τ_i.

Proof. The best-case completion time $BX^g_{i,j}$ of the j-th pure GPU segment of τ_i given in Eq. (4.7) is directly adopted from [22], the detailed proof of which can be found in

that paper. Next, we determine the minimum number of higher-priority jobs of τ_h that can fully present in $BX^g_{i,j}$. Let us use $s^g_{i,j}$ and $c^g_{i,j}$ to denote the absolute start and completion of τ_i's j-th pure GPU segment. Assuming m arrivals of the higher-priority task τ_h during $BX^g_{i,j}$ with the start time of the m-th job $s_m \leqslant c^g_{i,j}$ and that of the first job $s_1 \geqslant s^g_{i,j}$, we can deduce that $c_{m-1} - s_1 \leqslant BX^g_{i,j}$ where c_{m-1} is the completion time of $m-1$-th job of τ_h, i.e., $c_{m-1} \leqslant s_m$. This indicates that there are at least $m-1$ jobs fully executed within $BX^g_{i,j}$. We can compute m using

$$\lfloor \frac{BX^g_{i,j}}{T_h} \rfloor,$$

and obtain the minimum number of fully overlapped jobs of τ_h, $m-1$, by

$$\lfloor \frac{BX^g_{i,j}}{T_h} \rfloor - 1.$$

Therefore, Eq. 4.6 gives the minimum amount of fully overlapped CPU execution. ■The overall minimum fully overlapped CPU execution of a higher-priority task τ_h for all pure GPU segments of task τ_i can be obtained by:

$$O^{cg}_{i,h} = \sum_{0<j\leqslant\eta^g_i} O^{cg}_{(i,j),h} \tag{4.8}$$

Eq. (4.8) computes the case for $\bigcup C_h \sqsubset G^e_{i,j}$. Similarly, for the overlapped execution case for $\bigcup G^e_h \sqsubset C_{i,j}$, we have:

$$O^{gc}_{i,h} = \sum_{0<j\leqslant\eta^c_i} O^{gc}_{(i,j),h} \tag{4.9}$$

where

$$O^{gc}_{(i,j),h} = \max((\lceil \frac{BX^c_{i,j}}{T_h} \rceil - 1) \cdot \widetilde{G^e_h}, 0) \tag{4.10}$$

Based on the above, we now derive improved analyses.

Lemma 20 *Under the IOCTL-based approach, the worst-case response time for a self-suspending task τ_i is bounded by:*

$$\begin{aligned} R_i &= C_i + G_i + B_i \\ &+ \sum_{\tau_h \in hpp(\tau_i)} (\lceil \frac{R_i + J_h^c}{T_h} \rceil (C_h + G_h^m) - O_{i,h}^{cg}) \\ &+ \sum_{\substack{\eta_i^g > 0 \wedge \eta_h^g > 0 \\ \wedge \tau_h \in hp(\tau_i)}} (\lceil \frac{R_i + J_h^g}{T_h} \rceil G_h^e - O_{i,h}^{gc}) \end{aligned} \tag{4.11}$$

Proof. As $O_{i,h}^{cg}$ given by Eq. (4.8) guarantees the minimum overlapped CPU execution of τ_h with τ_i's pure GPU execution, this portion can be safely deducted from the CPU preemption time of τ_h. Similarly, $O_{i,h}^{gc}$ can be deducted from the GPU preemption time. ■

Lemma 21 *Under the IOCTL-based approach, the worst-case response time of a busy-waiting task τ_i is bounded by:*

$$\begin{aligned} R_i &= C_i + G_i + B_i \\ &+ \sum_{\tau_h \in hpp(\tau_i)} (\lceil \frac{R_i}{T_h} \rceil (C_h + G_h) - (O_{i,h}^{cg} + O_{i,h}^{gc})) \\ &+ \sum_{\substack{\eta_i^g > 0 \wedge \eta_h^g > 0 \\ \tau_h \in hp(\tau_i) \\ \wedge \tau_h \notin hpp(\tau_i)}} (\lceil \frac{R_i + J_h^g}{T_h} \rceil G_h^e - O_{i,h}^{gc}) \end{aligned} \tag{4.12}$$

Proof. With self-suspension, $\tau_h \in hpp(\tau_i)$ preempts τ_i for its entire CPU and GPU segment execution. Hence, we can safely deduct both $O_{i,h}^{cg}$ and $O_{i,h}^{gc}$ from the preemption time. For τ_h on different cores, $O_{i,h}^{gc}$ can be deducted as in Eq. (4.11). ■

4.7 Evaluation

We conduct schedulability experiments to compare the proposed approaches against prior work. Then we break down our approaches and assess the impact of the GPU priority assignment and the improved analysis. Lastly, we demonstrate a case study on an Nvidia embedded platform.

4.7.1 Schedulability Experiments

We generated 1,000 random tasksets for each experimental setting based on the parameters in Table 4.2. The parameter selection is inspired by the prior work [56], with slight modifications to increase the system load. Based on the measurement in Sec. 4.7.2, we aggressively set ϵ to 1 ms for our approaches, while assuming zero overhead for previous work. For each task in a taskset, the number of tasks on each CPU is first chosen, and the utilization per CPU is generated based on the UUniFast algorithm [17]. Then for each task, its period and the number of GPU segments are uniformly randomized within the given range. Then the parameters for each segment are determined. Task priority is assigned by the Rate Monotonic (RM) policy.

Comparison With Prior Work

We first compare our proposed approaches with two well-known synchronization-based methods, MPCP [56] and FMLP+ [20]. Both methods offer suspension-aware and busy-waiting analyses, and we evaluate the proposed approaches against them. For our approaches, we use the improved analysis given in Sec. 4.6.2 with the GPU priority assign-

Parameters	Value
Number of CPUs	4
Number of tasks per CPU	[3, 6]
Ratio of GPU-using tasks	[40, 60] %
Utilization per CPU	[0.4, 0.6]
Task Period	[30, 500] ms
Number of GPU segments per task	[1, 3]
Ratio of GPU exec. to CPU exec. (G_i/C_i)	[0.2, 0.5]
Ratio of GPU misc. in GPU exec. (G_i^m/G_i)	[0.1, 0.3]
Runlist update cost (ϵ)	1 ms

Table 4.2: Parameters for taskset generation

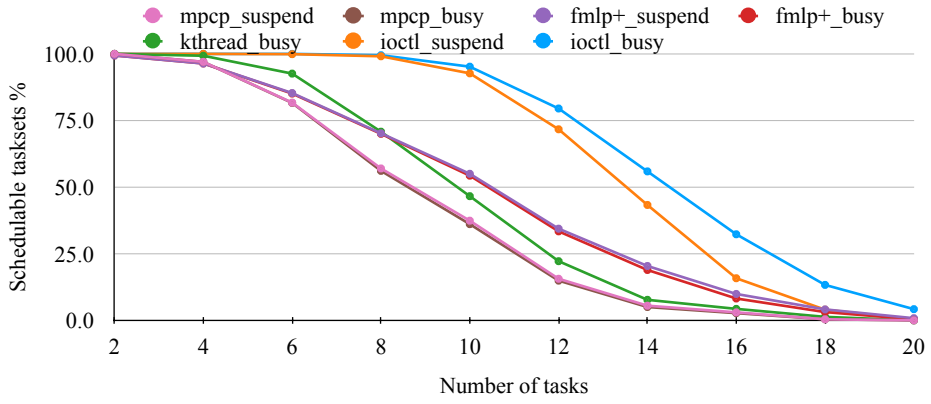


Figure 4.6: Schedulability w.r.t. the number of tasks

ment in Sec. 4.5.3. Hence, we first run the response time test for a taskset with the default RM priorities, and if the test fails, try again with separate priorities for GPU segments.

We investigate the impact of varying the number of tasks in the taskset, the number of CPUs, the utilization per CPU, and the ratio of GPU-using tasks in Figs. 4.6, 4.7, 4.8 and 4.9, respectively. The resutls show that, in general, the `ioctl_busy` and `ioctl_suspend` approaches outperform previous methods. However, the `kthread_busy` curve occasionally

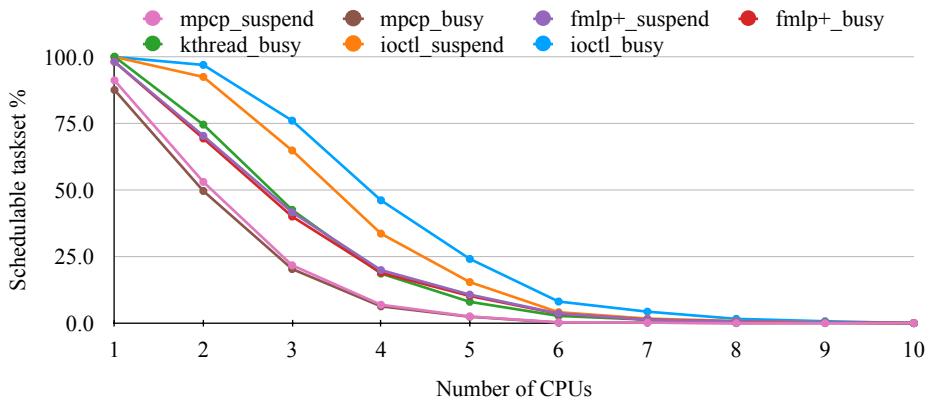


Figure 4.7: Schedulability w.r.t. the number of CPUs

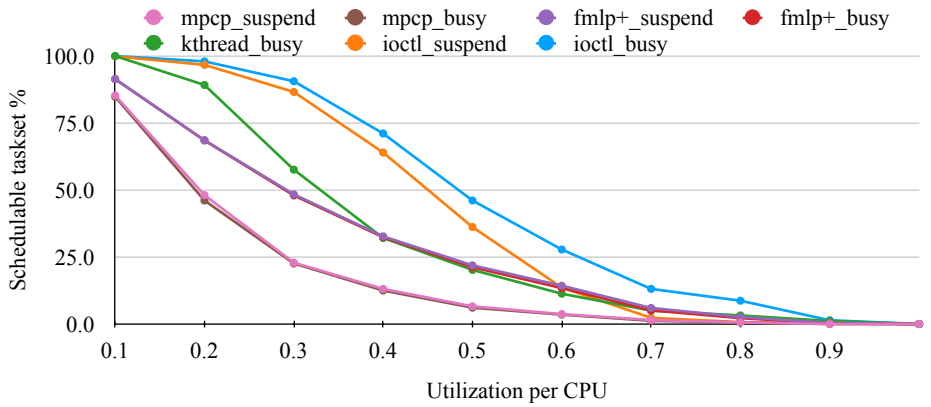


Figure 4.8: Schedulability w.r.t. the utilization per CPU

falls below those of previous methods due to its inherent drawback, e.g., as the utilization per CPU increases, the performance of `ioctl_busy` becomes worse than `fmlp+`. This is due to that `kthread_busy` cannot efficiently utilize computing resources as the system becomes increasingly loaded on the CPU side.

Fig. 4.10 examines the effect of changing the ratio of G_i/C_i. When $G_i/C_i = 0.1$, `ioctl_suspend` underperforms `fmlp+` but it does not continue as the ratio increases. This suggests that the benefits of `ioctl_suspend` become more visible as the ratio of GPU execution time (G_i) to CPU execution time (C_i) increases.

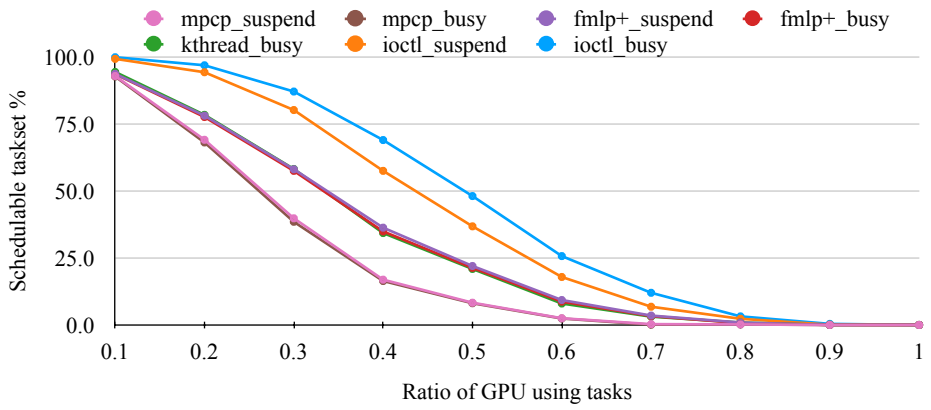


Figure 4.9: Schedulability w.r.t. the ratio of GPU-using tasks

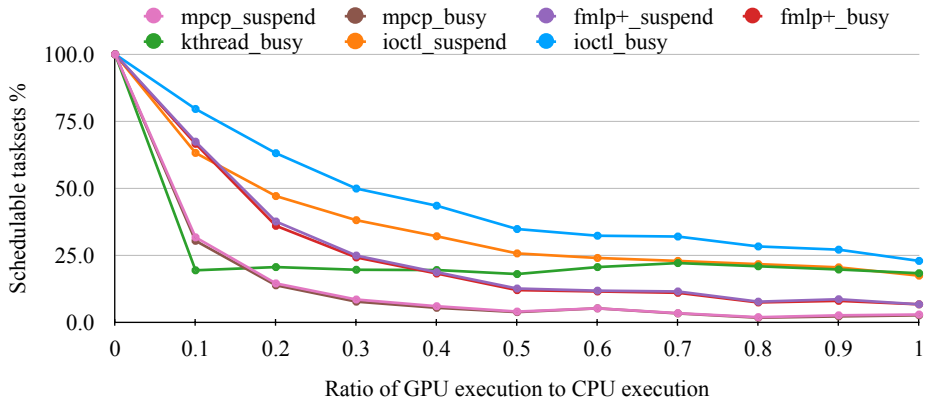


Figure 4.10: Schedulability w.r.t. the ratio of G_i to C_i

Lastly, we explore the impact of best-effort tasks running with the lowest priority in the system. After generating the tasks using the aforementioned method, we randomly designate a specific percentage of tasks as best-effort tasks in this experiment. Fig. 4.11 depicts the percentage of schedulable tasksets as the ratio of best-effort tasks increases. The rest of tasks are all real-time tasks in each taskset with constraint deadlines. The best-effort tasks contribute to blocking time in the analysis of `mpcp` and `fmlp+`. Since GPU preemption is enabled in our proposed approaches, they significantly outperform the prior methods.

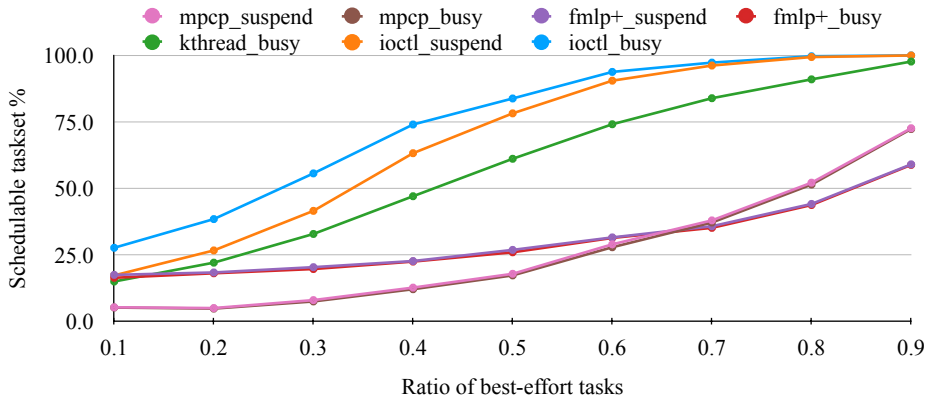


Figure 4.11: Schedulability w.r.t. the percentage of best-effort tasks

Effect of GPU Priority Assignment

In this experiment, we assess the impact of GPU priority assignment on taskset schedulability. We compare the performance under baseline analyses of `kthread_busy`, `ioctl_busy`, and `ioctl_suspend` with and without separate GPU priorities. The same taskset generation parameters in Table 4.2 are used here, and Fig. 4.12 depicts the gain that GPU priority assignment can bring about.

Effect of Improved Analysis

Next, we examine the impact of the improved analysis on the overall schedulability of the taskset. To enhance the likelihood of overlapped execution and amplify the effect, we use the following parameters to generate tasksets: (i) the number of CPUs is set to 2, (ii) two CPU tasks with high utilization and small periods and one GPU task with long GPU execution are always added to the taskset, and (iii) the number of tasks per CPU is [2, 4]. However, this limits our ability to experiment with adjusting the number of CPUs and the utilization per CPU, so we skipped these two configurations here. All other parameters

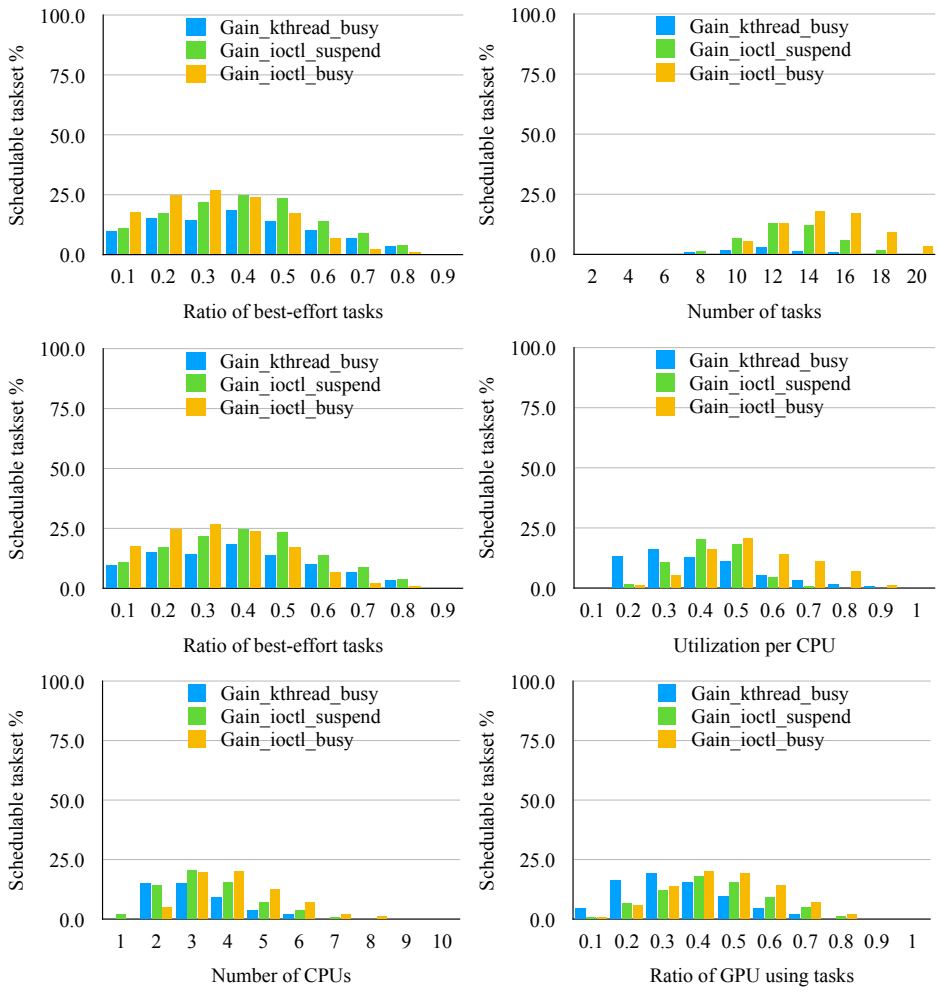


Figure 4.12: Improvement by GPU priority assignment

remain the same as in the previous experiments. Since the improvement cannot be applied to the kernel thread approach, we only compare the gain under the IOCTL-based approach, i.e., `ioctl_busy` and `ioctl_suspend`. The results are shown in Fig. 4.13.

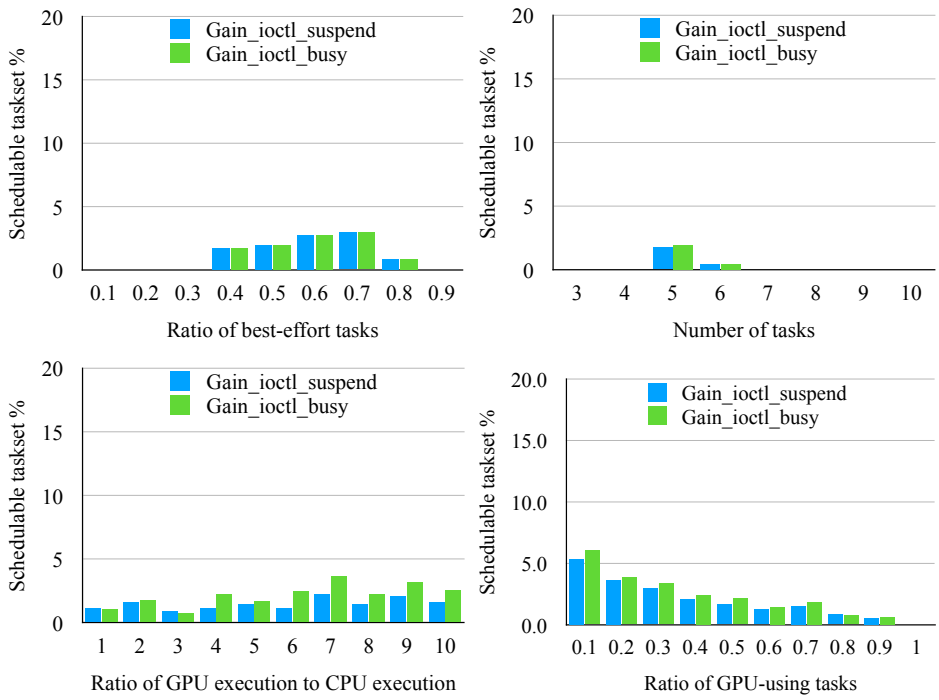


Figure 4.13: Improvement of schedulability analysis

4.7.2 System Evaluation

We implemented the proposed preemptive GPU scheduling approaches in L4T R35.1 with Jetpack 5.0.2 on an Nvidia Jetson Xavier NX Development Kit. This board is equipped with a 6-core 64-bit Carmel ARMv8.2 processor and we set it to operate at its maximum frequencies under 4-core 15W mode in the experiments.

Case Study. We conducted a case study on a real system to evaluate the performance and effectiveness of the proposed preemptive GPU scheduling mechanism. Table 4.4 provides a summary of the taskset employed in this study. The tasks in the table are arranged in descending order of priority. Tasks 2, 5, and 7 are CPU-only tasks with $G_i = 0$, while the remaining tasks involve GPU computations. Tasks 6, 7, and 8 are categorized as best-effort

	Max (us)	Min (us)	Avg (us)
Overhead	1036	333	476

Table 4.3: Runtime overhead of runlist update

tasks, as they are not assigned real-time priority. For `ioctl_suspend`, we used CUDA events with the `cudaEventBlockingSync` flag to suspend a task during its GPU execution. We compared our approaches along with `unmanaged`, which is the case for the default Nvidia GPU driver.

We executed the taskset for a duration of 1 minute and measured the worst-case observed response time for each real-time task. The results are depicted in Fig. 4.14. Under unmanaged GPU scheduling with interleaved execution, the response times become unpredictable, particularly for tasks 1 and 4. With the proposed approaches, task 5 exhibits high worst-case response time under both `ioctl_busy` and `kthread_busy`. This is because task 5 is a CPU task with the lowest real-time priority and the busy-waiting mode does not allow it to get CPU time while other higher-priority tasks are in GPU execution.

Overhead. While running the case study, we also measured the overhead ϵ of runlist update, and it is shown in Table 4.3. The measured overhead includes the cost of IOCTL system call, the cost of scheduling algorithms in kernel space, and the cost of runlist update. The maximum observed overhead of about 1 ms in our setup exceeds the range reported in prior work [24]. We suspect this is due to the relatively lower frequency of our GPU and it could be optimized in future generations of hardware. Nonetheless, we consider the cost acceptable based on our schedulability experiments with the same amount of overhead.

Task	Workload	C_i	G_i	$T_i = D_i$	CPU	Priority
1	`histogram`	5	14	100	1	99
2	`mmul_cpu_1`	116	0	200	1	98
3	`dxtc`	17	37	250	0	97
4	`mmul_gpu_2`	14	90	400	1	96
5	`mmul_cpu_2`	66	0	500	0	95
6	`mmul_gpu_1`	2	49	250	0	0
7	`mmul_cpu_3`	66	0	800	1	0
8	`simpleTexture3D`	0	inf	-	2	0

Table 4.4: Taskset used in case study

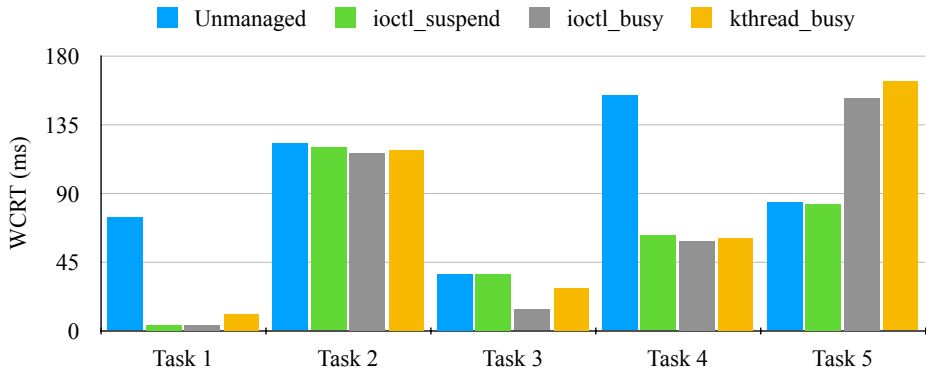


Figure 4.14: WCRT comparison between preemptive and unmanaged GPU scheduling

4.8 Conlusion

In this work, we have proposed two approaches: the IOCTL-based approach and the kernel thread approach to enable preemptive priority-based scheduling of GPU-using tasks in real-time systems. We first discussed how the Tegra GPU driver works and presented the design of our approaches. Then we provided a comprehensive response time analysis for these approaches, with an improvement for the IOCTL-based approach to reduce pessimism by considering the overlaps between different task segments using different computing resources. Through empirical evaluations and case studies, we have demonstrated the effectiveness of the proposed approaches in enhancing schedulability, and analyzed the

breakdown of the improvements made by the reduced pessimism analysis and GPU priority assignment. Additionally, our case study shows the benefits of our approaches in predictability and responsiveness over the default GPU driver policy.

Future work can focus on further optimizing and refining the proposed approaches and exploring additional scheduling strategies such as dynamic priority. Combing our device-driver level approaches with GPU partitioning mechanisms will also be an interesting direction.

Chapter 5

Conclusions

5.1 Summary of the Contributions

In this dissertation, we have addressed the challenges in real-time GPU scheduling through the development of novel algorithms and system-level solutions.

One of the key contributions is the introduction of the sBEET framework, which explores GPU spatial multitasking to achieve a balance between energy efficiency and schedulability. Our analysis shows that the active idle SMs causes energy inefficiency, and through extensive analysis and optimization, the sBEET framework offers better system performance in meeting tasks' deadlines as well as reducing energy consumption.

Additionally, we have proposed the sBEET-mg framework built upon sBEET specifically designed for multi-GPU systems, to enhance both energy efficiency and schedulability. This framework incorporates offline task distribution and runtime job migration strategies, effectively distributing computational workloads across multiple GPUs and optimizing resource allocation in real-time applications.

Furthermore, we have enabled preemptive priority-based scheduling for real-time GPU tasks, introducing two approaches to manage preemption. Through comprehensive analysis with reduced pessimism, our proposed approaches outperform sync-based approaches, providing improved responsiveness and enhanced schedulability.

Overall, this dissertation contributes to the field of real-time GPU scheduling by proposing innovative frameworks, algorithms, and analysis techniques, offering insights into energy efficiency, schedulability, and preemptive scheduling for real-time GPU tasks. These contributions pave the way for further advancements in real-time systems and GPU scheduling techniques.

5.2 Future Research Directions

One promising direction for future research is to further explore energy-aware preemptive scheduling techniques. This involves integrating energy considerations into the scheduling framework while leveraging the capabilities of GPU preemption. The objective is to minimize energy usage without compromising real-time constraints. This can be achieved through the development of advanced energy models and optimization algorithms that dynamically adapt the scheduling decisions based on both task characteristics and energy constraints. The research in this area can contribute to more energy-efficient real-time systems, making them more sustainable and environmentally friendly.

Another area for future exploration is the combination of GPU preemption and task partitioning techniques. This research direction aims to provide fine-grained control over resource allocation and scheduling in real-time GPU systems. By effectively leveraging

the capabilities of GPU preemption alongside task partitioning strategies, it becomes possible to achieve optimized resource utilization and improved system performance. The focus can be on developing novel algorithms and mechanisms that dynamically allocate resources based on task characteristics, workload conditions, and system constraints. This research has the potential to enhance the scalability and efficiency of real-time GPU systems in diverse application domains.

Dynamic priority assignment algorithms offer an avenue for improving task scheduling and resource utilization in real-time GPU systems. Future research can focus on developing adaptive priority assignment techniques that take into consideration not only real-time constraints but also the dynamic nature of the system. This involves designing algorithms that can dynamically assign priorities to tasks based on their urgency, computational requirements, and system conditions. The goal is to achieve optimal task scheduling and resource utilization while ensuring that critical real-time tasks receive the necessary attention and resources. This research direction has the potential to enhance the responsiveness and performance of real-time GPU systems, especially in scenarios with varying workload conditions and resource availability.

Bibliography

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

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

[7] Mohammad Abdel-Majeed, Daniel Wong, and Murali 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.

[8] 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.

[9] 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.

[10] 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.

[11] AnandTech. The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review. https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review.

[12] Neil C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. 2007.

[13] Joshua Bakita and James H. Anderson. Hardware Compute Partitioning on NVIDIA GPUs. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2023.

[14] 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.

[15] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Transactions on parallel and distributed systems*, 20(4):553–566, 2008.

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

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

[18] Konstantinos Bletsas, Neil C. Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. *Leibniz Transactions on Embedded Systems*, 5(1):02:1–02:20, May 2018.

[19] Björn B Brandenburg. The FMLP+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 61–71. IEEE, 2014.

[20] Björn B. Brandenburg. The fmlp+: An asymptotically optimal real-time locking protocol for suspension-aware analysis. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 61–71, 2014.

[21] Robert A. Bridges, Neena Imam, and Tiffany M. Mintz. Understanding GPU power. a survey of profiling, modeling, and simulation methods. *ACM Computing Surveys*, 49(3), 9 2016.

[22] Reinder Bril, Elisabeth Steffens, and Wim Verhaegh. Best-case response times and jitter analysis of real-time tasks. *J. Scheduling*, 7:133–147, 03 2004.

[23] Martin Burtscher, Ivan Zecena, and Ziliang Zong. Measuring GPU power with the K20 built-in sensor. In *Proceedings of Workshop on General Purpose Processing Using GPUs*, GPGPU-7, page 28–36, New York, NY, USA, 2014. Association for Computing Machinery.

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

[25] 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.

[26] Guoyang Chen, Yue Zhao, X. Shen, and Huiyang 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.

[27] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang. Run-time technique for simultaneous aging and power optimization in GPGPUs. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

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

[29] Glenn Elliott and James Anderson. An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems. *Real-Time Systems*, 49(2):140–170, 2013.

[30] Glenn Elliott et al. GPUSync: A framework for real-time GPU management. In *IEEE Real-Time Systems Symposium (RTSS)*, 2013.

[31] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, Carlsbad, CA, July 2022. USENIX Association.

[32] Sunpyo Hong and Hyesoon Kim. An integrated GPU power and performance model. *ACM SIGARCH Computer Architecture News*, 38:280, 2010.

[33] Seyedmehdi Hosseinimotlagh, Ali Ghahremannezhad, and Hyoseung Kim. On dynamic thermal conditions in mixed-criticality systems. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 336–349. IEEE, 2020.

[34] Seyedmehdi Hosseinimotlagh and Hyoseung 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.

[35] Seyedmehdi Hosseinimotlagh and Hyoseung 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.

[36] 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.

[37] Ali Jahanshahi, Hadi Zamani Sabzi, Chester Lau, and Daniel Wong. GPU-NEST: Characterizing energy efficiency of multi-GPU inference servers. *IEEE Computer Architecture Letters*, 19(2):139–142, 2020.

[38] 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.

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

[40] Yunji Kang, Woohyun Joo, Sungkil Lee, and Dongkun Shin. Priority-driven spatial resource sharing scheduling for embedded graphics processing units. *Journal of Systems Architecture*, 76:17–27, 2017.

[41] Mohsen Karimi, Yidi Wang, and Hyoseung 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.

[42] 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.

[43] Shinpei Kato, Carnegie Mellon University, The University of Tokyo, Karthik Lakshmanan, Ragunathan Rajkumar, and Yutaka Ishikawa. Timegraph: GPU scheduling for real-time multi-tasking environments. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, Portland, OR, 2011. USENIX Association.

[44] 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)*, pages 1–10, 2017.

[45] 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.

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

[47] Youngmoon Lee, Kang G Shin, and Hoon Sung Chwa. Thermal-aware scheduling for integrated CPUs-GPU platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–25, 2019.

[48] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Kim, Tor Aamodt, and Vijay Janapa Reddi. Gpuwattch: enabling energy optimizations in gpgpus. *ACM SIGARCH Computer Architecture News*, 41, 07 2013.

[49] 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.

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

[51] Nordic Semiconductor. Nrf52832 soc. `https://www.nordicsemi.com/products/nrf52832`.

[52] 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.

[53] Nathan Otterness and James H Anderson. AMD GPUs as an alternative to NVIDIA for supporting real-time workloads. In *Euromicro Conference on Real-Time Systems (ECRTS)*, 2020.

[54] Nathan Otterness, Ming Yang, Tanya Amert, J Anderson, and F Donelson Smith. Inferring the scheduling policies of an embedded CUDA GPU. OSPERT, 2017.

[55] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H Anderson, F Donelson Smith, Alex Berg, and Shige 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.

[56] Pratyush Patel, Iljoo Baek, Hyoseung Kim, and Ragunathan Rajkumar. Analytical enhancements and practical insights for MPCP with self-suspensions. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[57] Ragunathan Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings., 10th International Conference on Distributed Computing Systems*, pages 116–117. IEEE Computer Society, 1990.

[58] 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.

[59] 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.

[60] Qingxiao Sun, Yi Liu, Hailong Yang, Zhongzhi Luan, and Depei 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.

[61] 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.

[62] Zois-Gerasimos Tasoulas and Iraklis Anagnostopoulos. Improving GPU performance with a power-aware streaming multiprocessor allocation methodology. *Electronics*, 8(12), 2019.

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

[64] A. M. Van Tilborg and G. M. Koob. Foundations of real-time computing: Scheduling and resource management. 141, 2012.

[65] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on GPUs. *TACO*, 8:13, 10 2011.

[66] 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.

[67] Yidi Wang, Mohsen Karimi, and Hyoseung Kim. Towards Energy-Efficient Real-Time Scheduling of Heterogeneous Multi-GPU Systems. In *2022 IEEE Real-Time Systems Symposium (RTSS)*, pages 409–421. IEEE, 2022.

[68] Yidi Wang, Mohsen Karimi, Yecheng Xiang, and Hyoseung Kim. Balancing energy efficiency and real-time performance in GPU scheduling. In *2021 IEEE Real-Time Systems Symposium (RTSS)*, pages 110–122. IEEE, 2021.

[69] Yidi Wang and Hyoseung 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.

[70] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[71] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. Quality of service support for fine-grained sharing on gpus. pages 269–281, 06 2017.

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

[73] Yecheng Xiang and Hyoseung 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.

[74] Houssam Eddine Zahaf, Abou Benyamina, Richard Olejnik, and Giuseppe Lipari. Energy-efficient scheduling for moldable real-time tasks on heterogeneous computing platforms. *Journal of Systems Architecture*, 74, 01 2017.

[75] 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.

[76] An Zou, Jing Li, Christopher D Gill, and Xuan Zhang. RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization. *IEEE Transactions on Parallel and Distributed Systems*, 2023.