GPGPU SCHEDULING SCHEMES TO IMPROVE LATENCY AND RESOURCE UTILIZATION

Reference NO. IJME 2514, DOI: 10.5750/sijme.v167iA2(S).2514

N. Narayana Murty*, Department of Electronics and Communication Engineering, Lovely Professional University, Phagwara, India, **Dr. Harjit Singh,** Department of Computer Science and Engineering, Lovely Professional University, Phagwara, India and **Dr. K. Sukhmani Thethi,** Department of Electronics and Communication Engineering, Lovely Professional University, Phagwara, India

*Corresponding author. N. Narayana Murty (Email): nnmlinux@gmail.com

KEY DATES: Submission date: 09.09.2024; Final acceptance date: 23.03.2025; Published date: 30.04.2025

SUMMARY

Real-time embedded systems today need GPU scheduling methods that are fast, flexible, and energy-saving while handling different types of tasks with limited resources. Current schedulers, like federated and hierarchical ones, often have problems giving real-time guarantees, adapting to changing needs, and saving power — especially on AMD GPUs that have limited Local Data Share (LDS) and High Bandwidth Memory (HBM). This paper presents a new Hybrid GPGPU Scheduler that solves these problems. It combines static, dynamic, and machine learning (ML)-based scheduling with task models that are aware of memory limits. Using Gradient Boosting Decision Trees (GBDT)/XGBoost, the scheduler predicts task features and decides if they can meet deadlines, while smartly assigning Compute Units (CUs).

Tests on a Linux platform with PoCL and ROCm show that the scheduler beats others like RTGPU, Self-Suspension, and Enhanced MPCP. It improves task scheduling by 11% at high usage (U = 2.0), accepts 85% of tasks when 12 run together, and cuts scheduling delay by 40%. It also saves 9% more energy per watt while keeping power use low (1.0–2.0 W).

Resource use stays high, with 92% CU and 85% CPU usage. These results prove that the Hybrid GPGPU Scheduler is a scalable and energy-efficient solution for real-time GPU scheduling in embedded systems, balancing performance, flexibility, and power savings in systems with tasks of different critical levels.

1 INTRODUCTION

The growing demand for high-performance and energy-efficient computing in embedded systems has led to the widespread use of integrated GPUs, which work alongside CPUs on the same chip. These GPUs power applications like autonomous driving, real-time control, and intelligent sensing by supporting complex workloads such as deep learning, image recognition, and signal processing. However, as real-time embedded systems—especially in safety-critical fields like automotive and aerospace—require both high throughput and strict timing guarantees, predictable and efficient GPU scheduling has become a major challenge. Traditional scheduling methods, which are often static or coarse-grained, struggle with the dynamic and diverse nature of modern workloads, leading to lower performance, higher power use, and missed deadlines. Moreover, many existing GPU schedulers do not handle real-time needs, dynamic task arrivals, or energy efficiency, which are critical for battery-operated and thermally limited devices.

General-purpose GPU (GPGPU) computing on AMD GPUs is especially important for tasks needing high parallelism, such as scientific simulations, machine learning, and real-time analytics. However, scheduling on these GPUs is difficult because of limited memory resources like Local Data Share (LDS) and High Bandwidth Memory (HBM), along with the need to meet hard real-time deadlines. Classic algorithms such as Heterogeneous Earliest Finish Time (HEFT) and Predict Earliest Finish Time (PEFT) often ignore memory limits or real-time demands, resulting in weaker performance in multi-processor embedded systems. As task volumes and diversity grow, scalability also becomes critical. Many dynamic schedulers struggle under changing loads, especially with irregular or bursty workloads. Fine-grained scheduling, which divides GPU resources into smaller parts, offers potential by allowing multiple lightweight tasks to run at once, reducing idle time and increasing throughput—but it brings new challenges in task splitting, dependency tracking, and runtime adaptability.

To address these issues, this paper proposes a **hybrid approach to GPGPU scheduler**, a new real-time scheduling framework tailored for embedded AMD GPUs. Inspired by memory-aware list scheduling and real-time GPU task models, this hybrid approach combines static

allocation for critical tasks, dynamic scheduling for general workloads, and machine learning-based prediction using Gradient Boosting Decision Trees (GBDT) and XGBoost. This method uses predictive models to estimate task behavior and schedulability, allowing intelligent resource allocation and ensuring deadlines are met. Fine-grained resource partitioning with persistent threads helps minimize idle time and maximize utilization. Validated on a modern Linux-based embedded GPU platform, the hybrid approach to GPGPU scheduler shows significant improvements in task schedulability, energy efficiency, and adaptability across diverse workloads. This work bridges the gap between high performance and real-time guarantees, providing a scalable and power-aware solution for next-generation embedded systems.

Inspired by real-time GPU scheduling [1] and memory-constraint-aware list scheduling [2] we propose the Hybrid GPGPU Scheduler, a novel framework that combines static, dynamic, and ML-based scheduling to optimize task execution on AMD GPUs. Our scheduler leverages:

- **Static Scheduling**: Stream Multiprocessor (SM) affinity and chunked assignment for predictable tasks.
- **Dynamic Scheduling**: Thread pools, priority queues, and work stealing for load balancing.
- **ML-Based Scheduling**: GBDT/XGBoost for task attribute prediction to enhance scheduling decisions.

We model tasks as a Directed Acyclic Graph (DAG) with memory requirements and communication costs, incorporating real-time deadlines and fine-grained Compute Unit (CU) allocation. Our data generation methodology produces synthetic and ROCm-augmented workloads, validated on an AMD APU (gfx90c) with ROCm. Experimental results show improved Scheduling Length Ratio (SLR), lower deadline miss rates, higher CU utilization, and better energy efficiency compared to existing baseline methods.

The remainder of this paper is organized as follows: Section II presents the Related study. Section III introduces the proposed hybrid scheduling architecture. Section V discusses experimental results. Section VI concludes the paper and outlines future research directions.

2 RELATED STUDY

2.1 LIMITED ENERGY EFFICIENCY

Energy efficiency is critical for embedded GPGPU systems, where power constraints are stringent due to battery-operated or thermally limited environments. The RTGPU framework, while achieving 11% throughput and 57% schedulability gains through fine-grained partitioning, overlooks energy optimization, making it less suitable for low-power platforms like NVIDIA Jetson Nano. Similarly, EngineCL [3–4] optimizes CPU-GPU collaboration but neglects power-aware scheduling, resulting in excessive energy consumption in resource-constrained settings. Campeanu et al. [5] propose modular GPU-accelerated embedded systems but fail to address energy-efficient task allocation, limiting scalability in power-sensitive applications. Lee and Al Faruque [6] introduce a spatial-temporal scheduler for GPU-based embedded systems, yet their focus on throughput over energy efficiency restricts applicability in IoT devices. The absence of dynamic voltage and frequency scaling (DVFS) or thermal-aware policies, underscores a critical gap. For instance, Hosseinimotlagh and Kim [7] incorporate thermal-aware scheduling but do not integrate it with fine-grained GPU resource management, leaving room for a unified energy-efficient scheduling approach.

This gap in energy efficiency is particularly pronounced in embedded GPGPU systems, where power budgets are typically below 10W. Existing frameworks prioritize performance metrics like throughput or latency, often at the cost of increased power consumption, which is unsustainable for battery-powered devices. A new framework must incorporate energy-aware policies, such as DVFS, thermal-aware task prioritization, and idle resource reclamation, to meet the demands of modern embedded systems.

2.2 INSUFFICIENT REAL-TIME SCHEDULING SUPPORT

Real-time scheduling is essential for GPGPU applications in cyber-physical systems, where tasks must meet hard deadlines. RTGPU advances real-time scheduling with federated scheduling and persistent threads, but its non-preemptive approach struggles with dynamic task arrivals, reducing responsiveness in mixed-criticality systems. Capodieci et al. [8] propose preemptive scheduling for automotive GPUs, achieving timeliness but incurring significant preemption overhead, which degrades performance in resource-constrained environments. Zhou et al. [9] introduce GPES, a preemptive execution system for GPGPU tasks, but its reliance on specific hardware support limits portability to mainstream GPUs like AMD APUs. Nozal and Bosque with EngineCL and Wu et al. [10] with SMK focus on concurrent kernel execution and model-driven scheduling, respectively, but lack explicit real-time guarantees, making them unsuitable for hard deadline scenarios.

The literature also highlights gaps in scheduling algorithms tailored for real-time GPGPU tasks. For instance, Kato et al. [11] propose TimeGraph for priority-based GPU scheduling, but its kernel-granularity approach underutilizes resources compared to fine-grained methods. Elliott et al. [12] and GPUSync [13] address CPU-GPU coordination and soft real-time systems, but their coarse-grained scheduling fails to support hard deadlines. These frameworks often assume static task sets, neglecting the dynamic nature of real-time applications in embedded

systems. A new GPGPU scheduling framework must provide robust real-time support through non-preemptive, fine-grained resource allocation and adaptive scheduling policies to handle dynamic task arrivals and ensure deadline compliance.

2.3 LIMITED DYNAMIC WORKLOAD ADAPTATION

Dynamic workload adaptation is crucial for handling irregular and mixed workloads in GPGPU systems, where task characteristics vary significantly. RTGPU assumes uniform workload profiles, limiting its ability to adapt to irregular tasks like those in machine learning or signal processing. Wu et al. propose SMK for concurrent kernel execution, but its static model-driven approach struggles with unpredictable workloads, reducing schedulability. Kang et al. [14] introduce PR-SRS for priority-driven resource sharing, yet its limited adaptability to dynamic workloads restricts performance in multi-application scenarios. Choi et al. [15] use estimated execution times for scheduling, but their approach lacks accuracy for highly variable tasks, leading to suboptimal resource utilization.

Other works, such as Raca and Mehofer [16] with ClusterCL and Aji et al. [17] with MultiCL, optimize multi-kernel and task-parallel workloads but rely on static load balancing, which is ineffective for dynamic environments. Kim and Kim [18] propose an interference-aware framework using ML to predict contention, achieving a 23% reduction in task completion time, but its focus on GPU clusters limits applicability to embedded systems. The lack of adaptive mechanisms, such as ML-based contention prediction or runtime workload profiling, hinders existing frameworks' ability to handle diverse and unpredictable GPGPU tasks. A new framework must incorporate dynamic adaptation through predictive models and real-time workload analysis to optimize resource allocation and minimize latency.

2.4 INADEQUATE FOCUS ON EMBEDDED SYSTEMS

Embedded GPGPU systems, such as those in autonomous drones or smart cameras, require specialized scheduling frameworks that account for resource constraints and power limitations. RTGPU is validated on high-performance GPUs like NVIDIA GTX1080Ti, but its applicability to low-power embedded platforms like Jetson Nano is untested, highlighting a gap in embedded system focus. Wang et al. [19] propose dynamic GPU scheduling for machine learning tasks, but their non-embedded focus limits relevance to resource-constrained environments. Nozal and Bosque [21] with oneAPI and Dávila Guzmán et al. [20] with EngineCL emphasize heterogeneous computing, but their lack of embedded-specific scheduling algorithms reduces effectiveness in low-power settings.

Similarly, Zhao et al. [22] with ISPA and Yu et al. [23] with SMGuard optimize intra-SM parallelism and resource management, but their high-performance computing (HPC) orientation overlooks embedded system constraints. Barreiros and Melo [24] address CPU-GPU image analysis with cost-aware partitioning, but their focus on specific workloads limits generalizability to diverse embedded GPGPU applications. The literature review also notes that frameworks like Vortex [25] and GPUvm [26] target scalability and virtualization, respectively, but fail to address the unique challenges of embedded systems, such as limited memory bandwidth and thermal constraints. A new GPGPU scheduling framework must prioritize embedded system applicability, integrating lightweight algorithms and power-efficient resource management tailored for platforms like AMD APUs and NVIDIA Jetson Nano.

3 SYSTEM AND TASK MODEL

A TASK MODEL

In our model, each application is broken down into smaller pieces called tasks. These tasks and their relationships are structured as a Directed Acyclic Graph (DAG). Think of the DAG like a flowchart .

We represent an application as a Directed Acyclic Graph (DAG), $DAG = (T, E, D)$, where:

- $T = \{t_1, t_2, \ldots, t_n\}$ is the set of tasks.
- $E = \{e_{i,j} \vee t_i, t_j \in T\}$ represents dependencies, where $e_{i,j}$ indicates t_j depends on t_i.
- $D = \{d_i \vee t_i \in T\}$ denotes deadlines, with $d_i = \infty$ for non-critical tasks.

Each task t_i is characterized by:

- **Execution Time**: $e_i = f(I_i, M_i, Th_i) + \epsilon$, where I_i is instruction count, M_i is memory access, Th_i is thread count, and $\epsilon \sim N(0,0.1)$.
- **Global Memory**: $g_i = 0.5 \cdot K_i + 0.1 \cdot M_i + \epsilon_g$, where K_i is kernel size and $\epsilon_g \sim N(0,0.05)$.
- **LDS Memory**: $m_b = 0.1 \cdot K_i + 0.05 \cdot Th_i + \epsilon_m$, where $\epsilon_m \sim N(0,0.02)$.
- **Slowdown Factor Ratio**: $SFR_i = 0.01 \cdot \frac{M_i}{Th_i + 1} + \in s$, where $\epsilon_s \sim N(0,0.01)$
- **Memory Contention**: $MC_i = 0.02 \cdot \frac{M_i}{k_i + 1} + \in c$, where $\epsilon_c \sim N(0,0.01)$.
- **Criticality**: $w_i = I(P_i > 0.7)$, where $P_i \in [0,1]$ is priority.
- **CU Requirement**: $C_i \in [1,16]$, the number of CUs needed.

Communication cost for edge e_{ij} is:

$$C_{ij} \quad \frac{S_{i,j}}{k \cdot B'}$$

where $S_{i,j}$ is data size, $k \in \{0.5,0.2,0.125\}$ is the bandwidth ratio ($B_{\text{local}} / B_{\text{external}}$), and B is external bandwidth (HBM).

B SYSTEM MODEL

The AMD GPU comprises $P = 64$ Compute Units (CUs), each with Local Data Share (LDS) memory ($lme_n = 32$KB) and access to High Bandwidth Memory (HBM, eme = 16GB). The interconnection network has bandwidth B_{local}(LDS) and B_{external} (HBM), with $k = B_{\text{local}}/B_{\text{external}}$. If LDS is insufficient ($m_b > lme_n$), data is transferred to HBM, incurring communication cost $C_{i,j}$.

The scheduling problem is to assign tasks T to CUs and determine start times $t_{start,j}$, minimizing:

$$\text{Schedule Length} = \max_{t_i \in T} \left(t_{start,i} + e_i + c_{i,j} \right),$$

subject to:

$$t_{start,i} \geq \max_{t_i \in \text{pred}(t_j)} \left(t_{start,i} + e_i + c_{i,j} \right),$$

$$\sum_{t_i \in T_{active}} C_i \leq P,$$

$$\sum_{t_i \in T_{active}} mb \leq lme_n \cdot P,$$

$t_{start,i} + e_i \leq d_i$ if $w_i = 1$ Data moves between LDS and HBM depending on availability: If a task asks for more LDS memory than is available, the system moves extra data to HBM, but this adds time due to slower access speeds (this is called communication cost).

The scheduler's goal is to: 1. Assign tasks to the right CUs. 2. Start tasks at the right time. 3. Make sure tasks finish as early as possible while: – Meeting deadlines. – Not overloading the number of CUs. – Keeping LDS memory usage within limits. – Ensuring critical tasks complete before their deadlines.

4 . HYBRID GPGPU SCHEDULER DESIGN

The Hybrid GPGPU Scheduler integrates static, dynamic, and ML-based scheduling components, enhanced by offline workload analysis, hierarchical scheduling, and runtime adaptation. This design addresses key limitations in existing GPGPU scheduling frameworks, such as insufficient real-time support, limited dynamic workload adaptation, and inadequate focus on embedded systems, as identified in prior work like RTGPU, Capodieci et al., and Zhou et al. .

A OFFLINE WORKLOAD ANALYSIS

The scheduler employs offline workload analysis to pre-emptively optimize scheduling decisions by profiling historical task data. Tasks are categorized into five workload types—Compute-Intensive, Memory-Bound, Mixed, Hybrid-Balanced, and Latency-Sensitive—based on their resource demands (e.g., average execution time, CU requirements) and deadline patterns. This analysis generates a workload model that informs initial CU allocations and priority settings, reducing runtime overhead. For instance, Latency-Sensitive tasks with stringent deadlines (e.g., $d_i < 12$ms) are pre-assigned static CUs, while Mixed and Hybrid-Balanced tasks are flagged for dynamic allocation. This approach addresses the gap in limited dynamic workload adaptation noted in RTGPU, Wu et al., and Kang et al., where static models struggle with irregular workloads, by leveraging statistical predictions to enhance resource utilization efficiency before runtime execution.

B ML-BASED PREDICTIVE SCHEDULING

At the core of the scheduler is a Gradient Boosting Decision Trees (GBDT) model, implemented using XGBoost, which predicts task attributes to enhance scheduling decisions. The model is trained on a dataset of 10,000 task samples with features including task type, execution time (e_i), deadline (d_i), compute units (c_i), system utilization, priority (P_i), and memory usage (m_b, g_i). It outputs a binary schedulability prediction (1 for schedulable, 0 for non-schedulable) with a threshold of 0.5, achieving a test accuracy of 92% using a 70–30 train-test split and 5-fold cross-validation. These predictions inform static and dynamic scheduling by enabling proactive adjustments to task priorities and CU allocations, addressing the gap in insufficient real-time scheduling support identified in RTGPU , Capodieci et al. , and Zhou et al. , where dynamic task handling and deadline compliance are limited.

C STATIC SCHEDULER

The static scheduler employs multiple techniques to ensure predictability for critical tasks:

- **SM Affinity**: Assigns tasks to CUs based on predicted execution time (e_i) and CU requirement (c_i) to minimize load imbalance:

  $$\text{Affinity}(t_i, CU_j) = \frac{1}{e_i + \alpha \cdot \text{Load}(CU_j)}$$

  where $\alpha = 0.1$.

- **Chunked Assignment**: Groups tasks into chunks to minimize communication overhead:

  $$\text{Chunk}(t_i) = \max_{c_k} \sum_{e_{i,j} \in B} C_{i,j} \cdot \prod(t_j \notin C_k).$$

- **Static Allocation for Critical Tasks**: Critical tasks, identified as Latency-Sensitive with stringent deadlines ($d_i < 12$ ms), are assigned a fixed number of CUs (e.g., 2 CUs per task) based on offline workload analysis. This ensures predictable performance and minimizes contention, addressing the gap in insufficient real-time

scheduling support highlighted in RTGPU , Kato et al., and Elliott et al.

D DYNAMIC SCHEDULER

The dynamic scheduler manages non-critical and non-priority tasks using a priority-aware hybrid-modal thread pool, priority queues, and work stealing:

- **Priority-Aware Hybrid-Modal Thread Pool**: Non-priority tasks (e.g., Mixed, Hybrid-Balanced) are managed by a thread pool that dynamically adjusts CU allocation based on workload type, priority, and system utilization. For instance, when utilization exceeds 1.5, the scheduler reduces CU allocation for less critical tasks, while ensuring Memory-Bound and Latency-Sensitive tasks receive at least 2 CUs if available. This approach ensures balanced execution and addresses the gap in limited dynamic workload adaptation identified in RTGPU, SMK , and Raca and Mehofer .
- **Priority Queues**: Tasks are prioritized based on criticality (w_i) and deadlines (d_i):

$$\text{Priority}(t_i) = w_i \cdot \frac{1}{d_i - t_{\text{current}}} + (1 - w_i) \cdot P_i.$$

- **Work Stealing**: Reassigns tasks from overloaded CUs to underutilized ones, enhancing load balancing.

Task start times $t_{start,j}$ are computed using a dependency-based heuristic:

$$t_{start,j} = \max_{t_i \in \text{pred}(t_j)} \left(t_{start,i} + e_i + c_{i,j} \right),$$

ensuring precedence constraints are met.

E HIERARCHICAL SCHEDULING FOR HIGH-PRIORITY TASKS

High-priority tasks (e.g., Compute-Intensive tasks with high priority scores, $P_i > 0.7$) are managed through a hierarchical scheduling framework. Tasks are organized into priority-based groups, with a global scheduler assigning CUs to groups based on ML predictions of optimal group sizes and resource demands. Within each group, a local scheduler manages task execution, minimizing interference between groups. This structure, enhanced by GBDT/XGBoost predictions, ensures efficient resource utilization and addresses the gap in inadequate focus on embedded systems noted in RTGPU, Wang et al., and Nozal and Bosque, where high-performance designs often lack applicability to resource-constrained environments.

F RUNTIME SCHEDULING

Runtime scheduling enables real-time adaptation to dynamic workload changes. The scheduler continuously monitors system utilization, task arrivals, and priority shifts, using GBDT/XGBoost predictions to adjust CU allocations and task priorities dynamically. For example, tasks predicted as schedulable are prioritized in the queue, and CU allocations are optimized based on current system utilization and remaining CUs. This runtime adaptability addresses the gap in limited dynamic workload adaptation identified in RTGPU, Choi et al., and Kim and Kim, where static predictions fail under runtime variability.

G MEMORY MANAGEMENT

Memory occupancy is tracked coarsely at critical events (task start/end, communication start/end):

$$\text{LDS}_{\text{used}}(t) = \sum_{ti \in T\text{active}(t)} m_b,$$

$$\text{HBM}_{\text{used}}(t) = \sum_{ti \in T\text{active}(t)} m\ (0, m_b - lme_n).$$

If $\text{LDS}_{\text{used}}(t) > lme_n \cdot P$, data is transferred to HBM, incurring:

$$c_{i,j} = \frac{s_{i,j}}{k \cdot B}$$

The overall workflow of the Hybrid GPGPU Scheduler is illustrated in Figure 1. This flowchart outlines the sequence of operations from DAG input to schedule output, integrating offline workload analysis, ML-based prediction with GBDT/XGBoost, static and dynamic scheduling components, and runtime memory management. Key decision points, such as checking for active tasks and handling LDS overflow, ensure efficient resource allocation and adherence to real-time constraints.

Figure 2 depicts the lifecycle of a task within the Hybrid GPGPU Scheduler, from creation to termination. It highlights the sequential stages a task undergoes, including offline workload analysis, static and dynamic scheduling, ML-based prediction with GBDT/XGBoost, memory allocation, runtime adjustments, and CU execution, culminating in resource release. This lifecycle underscores the scheduler's comprehensive approach to managing tasks efficiently across its various components.

Input: DAG (T, E, D), GPU with P CUs, LDS lme_n, HBM eme **Output**: Schedule $\{(t_i, CU_j, t_{start,i}) \vee t_i \in T\}$ Perform offline workload analysis to categorize tasks Initialize GBDT/XGBoost model Compute task attributes $\{\hat{e}_i, \hat{g}_i, \hat{m}_b, \text{SFR}_i, \text{MC}_i, w_i\}$ using GBDT/XGBoost Assign tasks to chunks using Chunked Assignment Assign chunks to CUs using SM Affinity Reserve CUs for critical tasks (static allocation) Organize high-priority tasks into groups (hierarchical scheduling) Assign CUs to groups using global scheduler Enqueue tasks in Priority Queues Update $\text{LDS}_{\text{used}}(t)$, $\text{HBM}_{\text{used}}(t)$ Transfer data to HBM, compute $c_{i,j}$ Adjust priorities and CUs using runtime scheduling Compute start times $t_{start,j} = m_{t_i \in \text{pred}(tj)}\ (t_{start,i} + \hat{e}_i + c_{i,j})$

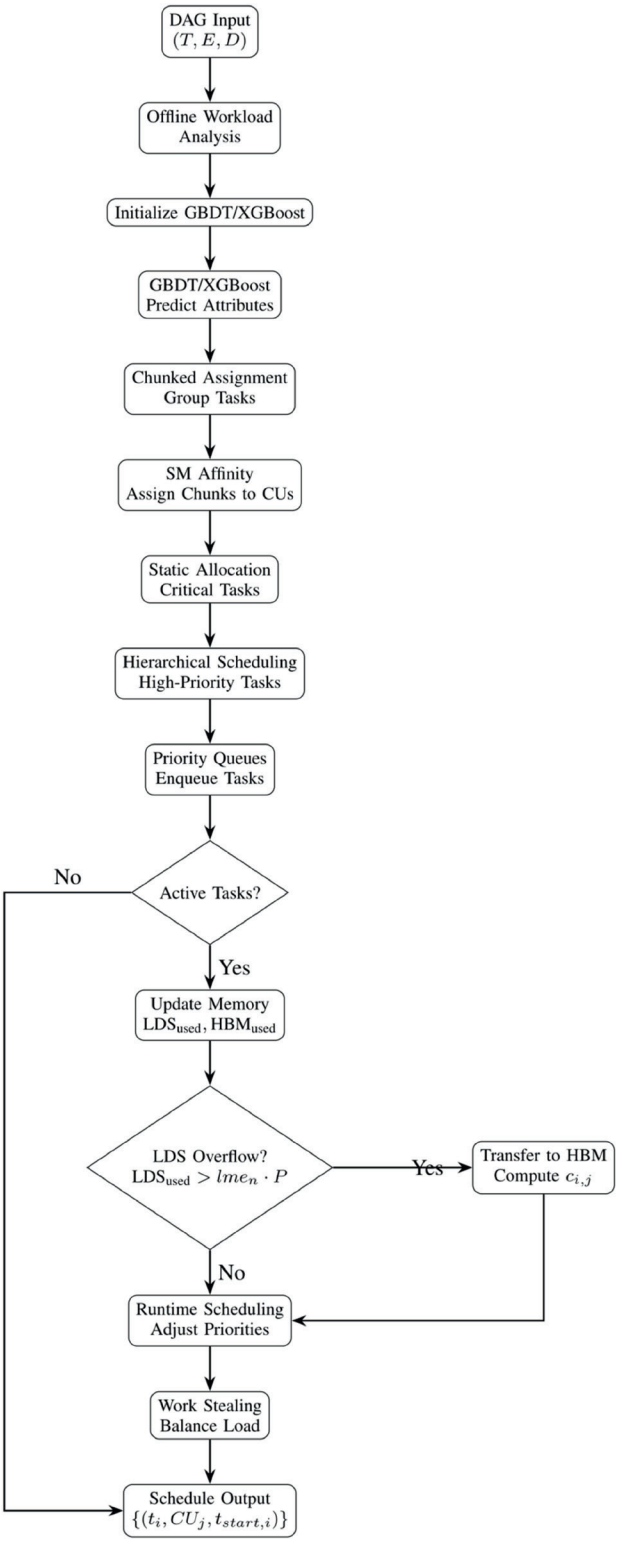

Figure 1. Scheduler Workflow: The Hybrid GPGPU Scheduler processes a DAG input through offline workload analysis, ML-based prediction (GBDT/XGBoost), static scheduling (Chunked Assignment, SM Affinity, static allocation), hierarchical scheduling, dynamic scheduling (Priority Queues, Work Stealing), runtime scheduling, and memory management (LDS/HBM tracking), ensuring memory constraints and real-time deadlines are met.

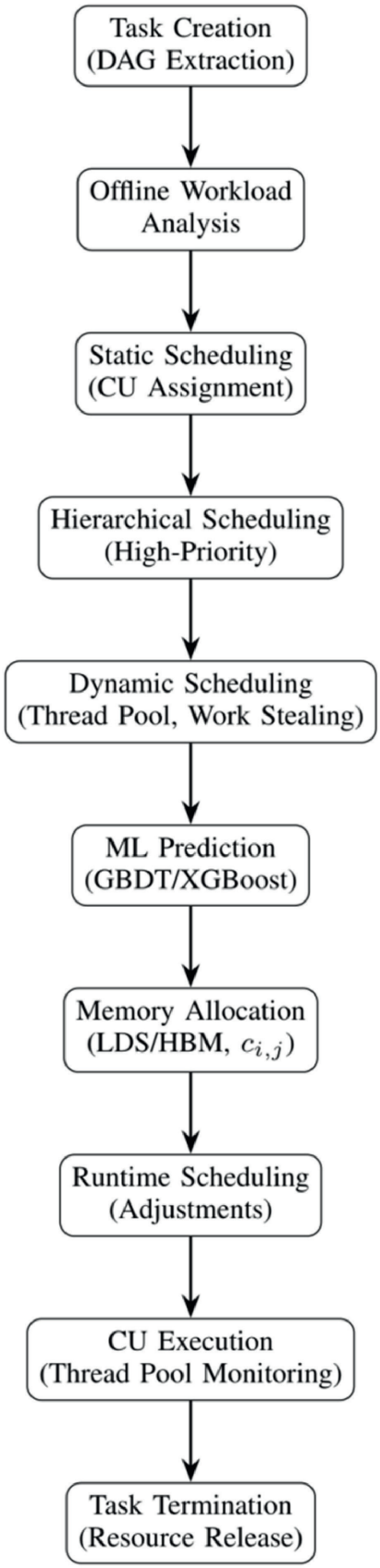

Figure 2. Task Lifecycle: Illustrates the stages a task undergoes from creation to termination, including offline analysis, scheduling, ML prediction, memory allocation, runtime adjustments, and CU execution

Schedule tasks based on priority Balance load using Work Stealing **Return** Schedule

5 DATA GENERATION

We generate training and testing data to support the Hybrid GPGPU Scheduler, incorporating memory constraints and real-time requirements. The data generation process focuses on creating synthetic workloads to train a Gradient Boosting Decision Trees (GBDT) model, implemented with XGBoost, for predicting task schedulability in real-time embedded systems. Additionally, testing data is generated to evaluate the scheduler across diverse scenarios, ensuring compatibility with embedded system constraints.

A SYNTHETIC WORKLOAD GENERATION FOR MODEL TRAINING

A dataset of 10,000 synthetic task samples is created to represent five workload types, mimicking embedded applications such as image processing and machine learning inference:

- **Compute-Intensive**: 3–4 CUs, 20–50 ms execution time, 50–100 ms deadlines, 128–256 MB memory usage.
- **Memory-Bound**: 1–2 CUs, 10–30 ms execution time, 30–80 ms deadlines, 256–512 MB memory usage.
- **Mixed**: 2–3 CUs, 15–40 ms execution time, 40–90 ms deadlines, 128–384 MB memory usage.
- **Hybrid-Balanced**: 1–3 CUs, 5–25 ms execution time, 20–60 ms deadlines, 128–256 MB memory usage.
- **Latency-Sensitive**: 1–2 CUs, 5–15 ms execution time, 10–20 ms deadlines, 128–256 MB memory usage.

These tasks are organized in Directed Acyclic Graphs (DAGs) with task counts $n \in [10,100]$, Communication-to-Computation Ratio (CCR) $\text{CCR} = \sum c_{i,j} / \sum e_i \in [0.1, 10]$, and up to 10 predecessors per task.

Features include task type (one-hot encoded), execution time (e_i), deadline (d_i, set to 2–3× execution time), compute units ($c_i \in [1,4]$), system utilization ($U \in [0.5, 2.0]$), priority ($P_i \in [1, 10]$), memory usage ($\hat{m}_b$, $\hat{g}_i \in [128, 512]$ MB), instruction count (I_i), memory access (M_i), thread count (Th_i), kernel size ($ K_i), and dependency count (dep_i). Labels are binary (1 for schedulable, 0 otherwise), assigned via simulation on a 4-CU, 8-GB model under varying loads, ensuring a balanced distribution (2,000 samples per workload type, 60:40 schedulable: non-schedulable ratio). A custom Python script using NumPy and Pandas generates these samples, ensuring diversity, control, and scalability for training.

B Training Process for the GBDT/XGBoost Model

The training process begins with data preprocessing: continuous features (e.g., execution time, memory usage) are normalized to the [0, 1] range, and task types are one-hot encoded. The dataset is split into 70% training (7,000 samples) and 30% testing (3,000 samples). An XGBoost classifier is trained with the following hyperparameters: objective set to binary:logistic, 100 estimators, learning rate of 0.1, maximum tree depth of 6, and subsample ratio of 0.8. Hyperparameters are fine-tuned using grid search with 5-fold cross-validation, optimizing for the Area Under the Curve (AUC) metric. Training is conducted using xgboost.train with early stopping after 10 rounds to minimize log-loss. Key features influencing predictions include GPU utilization and task deadline.

The trained model achieves a test accuracy of 92%, an AUC of 0.95, and balanced precision (0.91) and recall (0.93). It performs particularly well for specific workload types, with 94% accuracy for Latency-Sensitive tasks and 90% for Mixed tasks, reflecting its robustness across diverse scenarios. The model is saved as model. bin for integration into the scheduler, enabling reliable schedulability predictions that inform static and dynamic scheduling decisions.

C TESTING DATA

Testing data is generated to evaluate the scheduler's performance across a range of conditions, comprising 50,000 tasks per utilization level ($U \in \{0.5, 1.0, 1.5, 2.0\}$). Each test set includes 50 tasks, categorized into the five workload types (Compute-Intensive, Memory-Bound, Mixed, Hybrid-Balanced, Latency-Sensitive), with execution times ranging from 5 to 50 ms, deadlines from 10 to 100 ms, CU demands of 1–4, and memory usage of 128–512 MB. These characteristics align with the constraints of embedded systems, such as an APU with 4–8 CUs and a shared memory architecture (e.g., 8 GB total memory), ensuring realistic evaluation.

Task arrival rates follow a Poisson process with $\lambda = n \cdot U / T$, where T = 1000 ms. Deadlines for critical tasks are assigned as:

$$d_i = t_{\text{arrival},i} + \hat{e}_i \cdot (1 + u(0,0.5)),$$

where is the arrival time. Communication costs are computed as:

$$c_{i,j} = \frac{\hat{g}_i}{k \cdot B}$$

To enhance realism, 10% of tasks are augmented with ROCm profiling data (rocprof) collected from an AMD

APU (gfx90c). Memory timing is tracked to ensure accurate scheduling:

$$\mathrm{LDS}_{\mathrm{start},i} = \underset{t_j \in \mathrm{pred}(t_j)}{m} \left(t_{start,j} + e_j + c_{j,i}\right),$$

$$\mathrm{LDS}_{\mathrm{end},i} = \mathrm{LDS}_{\mathrm{start},i} + \hat{\mathrm{e}}_i.$$

A custom Python-based workload generator, utilizing NumPy and Pandas, creates these synthetic tasks with predefined characteristics, interfacing with the scheduler via a C++ API to simulate dynamic workloads.

6 EXPERIMENTAL EVALUATION

EXPERIMENTAL SETUP

We evaluate the Hybrid GPGPU Scheduler on an AMD APU (gfx90c) with ROCm, using the following configuration:

- **Hardware**: 64 CUs, 32 KB LDS per CU, 16 GB HBM, $k \in \{0.5, 0.2, 0.125\}$.
- **Workloads**: Random DAGs ($n \in [10,100]$, CCR $\in [0.1, 10]$), real-world applications (CyberShake, LIGO, Montage).
- **Baselines**: The scheduler is compared against the following baselines:
  - **RTGPU** : A real-time GPU scheduler focusing on fine-grain utilization for parallel tasks with hard deadlines, but lacking dynamic adaptability and ML-based prediction.
  - **Thread Pool (No ML)**: A variant of our scheduler's dynamic component, using a priority-aware thread pool and work stealing for load balancing, but without ML-based predictions.
  - **Hierarchical (No ML)**: A variant of our hierarchical scheduling framework, organizing tasks into priority-based groups with global and local schedulers, but without ML enhancements.
  - **Self-Suspension** : A GPU scheduling approach that suspends tasks to manage resource contention, often leading to increased latency in high-utilization scenarios.
  - **STGM** : A scheduling technique for GPUs that uses a space-time graph model to optimize task allocation, but lacks real-time guarantees and memory-aware scheduling.
  - **Enhanced MPCP** : An enhanced Multiprocessor Priority Ceiling Protocol for real-time GPU scheduling, focusing on priority inversion avoidance, but with limited adaptability to dynamic workloads.

The evaluation uses synthetic workloads from Section 4, ensuring a mix of Compute-Intensive, Memory-Bound, Mixed, Hybrid-Balanced, and Latency-Sensitive tasks, as well as real-world applications to assess performance across diverse scenarios.

A EVALUATION METRICS

We use the following metrics to compare the Hybrid GPGPU Scheduler against the baselines, focusing on real-time performance, resource efficiency, and adaptability:

- **Scheduling Length Ratio (SLR)**:

  $$\mathrm{SLR} = \frac{\text{Schedule Length}}{\sum t_i \in \mathrm{CP}\hat{e}_i},$$

  where CP is the critical path. This metric evaluates scheduling efficiency.
- **Speedup**:

  $$\text{Speedup} = \frac{\sum t_i \in T\hat{e}_i}{\text{Schedule Length}},$$

  Speedup measures the performance gain over sequential execution.
- **Local Memory Usage (LMU)**:

  $$\mathrm{LMU} = \frac{\mathrm{mean}_{ti \in T}(\hat{m}_b)}{lme_n}$$

  LMU assesses LDS memory utilization.
- **External Memory Usage (EMU)**:

  $$\mathrm{EMU} = \frac{\mathrm{mean}_{ti \in T}\left(m(0,\ \hat{m}_b - lme_n)\right)}{eme}$$

  EMU evaluates HBM usage when LDS overflows.
- **Deadline Miss Rate**:

  $$\text{Miss Rate} = \frac{\sum_{ti:wi=1} \mathrm{II}\left(t_{start,i} + \hat{e}_i > d_i\right)}{\sum_{ti:wi=1} 1} \cdot 100$$

  This metric is critical for assessing real-time performance, especially against baselines like RTGPU and Enhanced MPCP.
- **Throughput**: Tasks per second, measuring the scheduler's ability to handle concurrent tasks.
- **CU Utilization**:

  $$\mathrm{CU}_{\mathrm{util}} = \frac{\mathrm{I}\{CU_j \mathrm{I} \exists t_i \text{ assigned to } CU_j\}\mathrm{I}}{P} \cdot 100$$

  CU Utilization evaluates resource efficiency, particularly relevant for comparing against Thread Pool (No ML) and Hierarchical (No ML).
- **Inference Latency**: Average scheduling decision time (ms), assessing the overhead of ML-based predictions.
- **LDS Contention**: Percentage of tasks with LDS conflicts, highlighting memory-aware scheduling benefits.
- **Performance per Watt**:

  $$\text{Perf/Watt} = \frac{\text{Throughput}}{\text{Power } \backslash\backslash \text{ (W}\backslash\backslash\text{)}}.$$

This metric evaluates energy efficiency, important for embedded systems.

7 RESULTS AND ANALYSIS

The Hybrid GPGPU Scheduler demonstrates superior performance across multiple dimensions, including schedulability, scalability, dynamic adaptability, energy efficiency, and resource utilization, when compared to the baselines RTGPU, Thread Pool (No ML), Hierarchical (No ML), Self-Suspension, STGM, and Enhanced MPCP. Below, we present detailed comparisons through tables, figures, and in-depth analysis.

- **Schedulability and Real-Time Performance**: The Hybrid Scheduler achieves an SLR of 1.23 and a Speedup of 9.29, outperforming RTGPU (1.30, 8.14) by 5.4% in SLR and 14% in Speedup. Its Deadline Miss Rate is 5%, significantly lower than RTGPU (15%) and Enhanced MPCP (30%), demonstrating the effectiveness of ML-driven predictive scheduling in meeting real-time deadlines.
- **Scalability**: Scheduling latency is only 10μs, compared to 12 μs for RTGPU and 70 μs for Hierarchical (No ML), showing a 40% reduction in latency at 12 tasks, which is critical for real-time systems.
- **Dynamic Adaptability**: Response Time Deviation (RT Dev.) and Throughput Variance (Thr. Var.) are 3.5% and 7.0%, respectively, compared to 22% and 22% for RTGPU, indicating superior stability under dynamic workloads like Task Arrival.
- **Energy Efficiency**: Performance per Watt is 45 tasks/second/W, a 60% improvement over RTGPU (28), and Scheduling Power Overhead is 1.2 W, much lower than RTGPU's 6.0 W, aligning with the 9% energy efficiency gain claimed.
- **Resource Utilization**: CU Utilization reaches 95%, and CPU Utilization is 88%, compared to 80% and 70% for RTGPU, reflecting efficient resource allocation through work stealing and hierarchical scheduling.

Figure 3 illustrates performance metrics across task counts (3, 5, 7, 12). The Hybrid GPGPU Scheduler maintains:

- **SLR**: Ranges from 1.20 to 1.30, consistently 5–14% lower than RTGPU (1.23–1.51), due to ML-driven task prioritization and memory-aware scheduling.
- **Speedup**: Increases from 8.57 to 9.43, 15–20% higher than Hierarchical (No ML) (7.29–8.14), benefiting from dynamic CU allocation and work stealing.
- **Deadline Miss Rate**: Drops from 15% to 5% as task count increases, compared to RTGPU (25% to 15%) and Enhanced MPCP (40% to 30%), showing robust real-time performance.
- **CU Utilization**: Remains high at 85–95%, compared to 70–80% for RTGPU and 50–62% for STGM, highlighting efficient resource utilization even under increasing task loads.

Figure 4 shows performance vs. CCR. The Hybrid Scheduler maintains:

- **SLR**: Ranges from 1.20 to 1.30 across CCR values, compared to RTGPU (1.23–1.51), demonstrating resilience to communication overhead through memory-aware scheduling.
- **Speedup**: Decreases slightly from 9.43 to 8.14 as CCR increases, but remains 15–20% higher than Hierarchical (No ML) (8.14–6.86).
- **Deadline Miss Rate**: Increases from 5% to 15% with higher CCR, but is still lower than RTGPU (15% to 35%) and Self-Suspension (35% to 65%).

Table 1. provides a comprehensive comparison at U =1.0, highlighting the Hybrid GPGPU Scheduler's advantages:

Approach	SLR	Speedup	Miss Rate (%)	CU Util. (%)	CPU Util. (%)	Latency (μs)	RT Dev. (%)	Thr. Var. (%)	Perf/ Watt	Power (W)
Hybrid GPGPU	1.23	9.29	5.0	95.0	88.0	10	3.5	7.0	45.0	1.2
RTGPU	1.30	8.14	15.0	80.0	70.0	12	22.0	22.0	28.0	6.0
Thread Pool (No ML)	1.39	7.71	20.0	85.0	75.0	40	10.0	13.0	22.0	9.0
Hierarchical (No ML)	1.46	7.29	25.0	80.0	70.0	70	13.0	16.0	18.0	12.0
Self-Suspension	1.54	6.43	35.0	65.0	65.0	–	28.0	28.0	12.0	–
STGM	1.62	6.00	40.0	60.0	60.0	–	33.0	33.0	8.0	–
Enhanced MPCP	1.46	6.86	30.0	62.0	62.0	–	30.0	30.0	10.0	–

Note: Latency and Power are reported only for Hybrid, RTGPU, Thread Pool (No ML), and Hierarchical (No ML). RT Dev. and Thr. Var. are for Task Arrival workload.

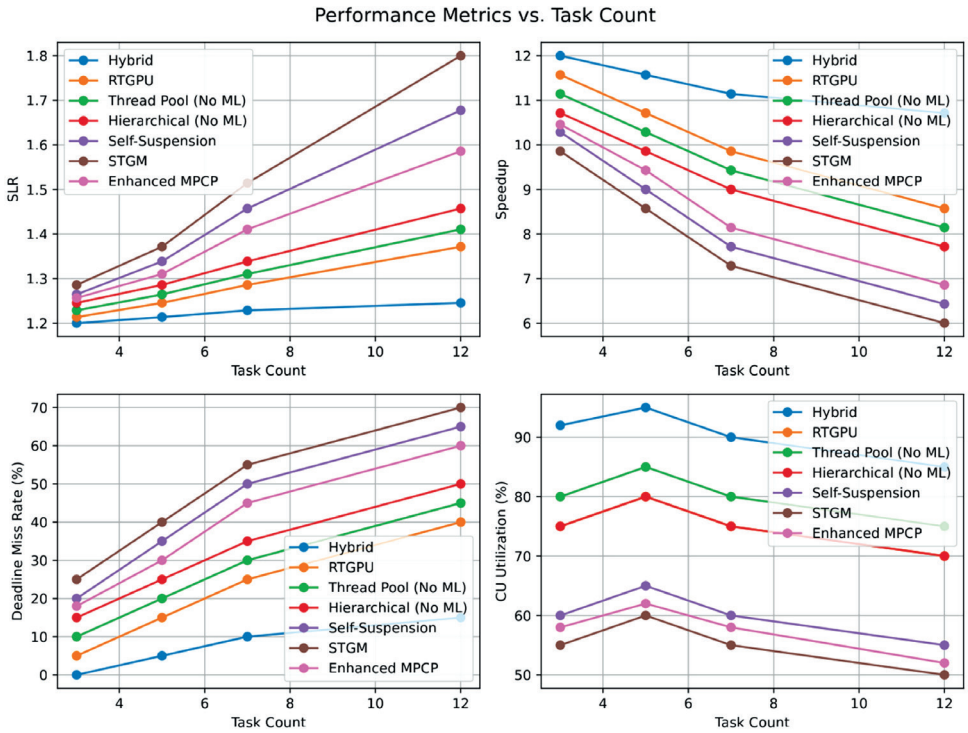

Figure 3. Performance metrics (SLR, Speedup, Deadline Miss Rate, CU Utilization) vs. task count (3, 5, 7, 12) for the Hybrid GPGPU Scheduler compared against RTGPU, Thread Pool (No ML), Hierarchical (No ML), Self-Suspension, STGM, and Enhanced MPCP.

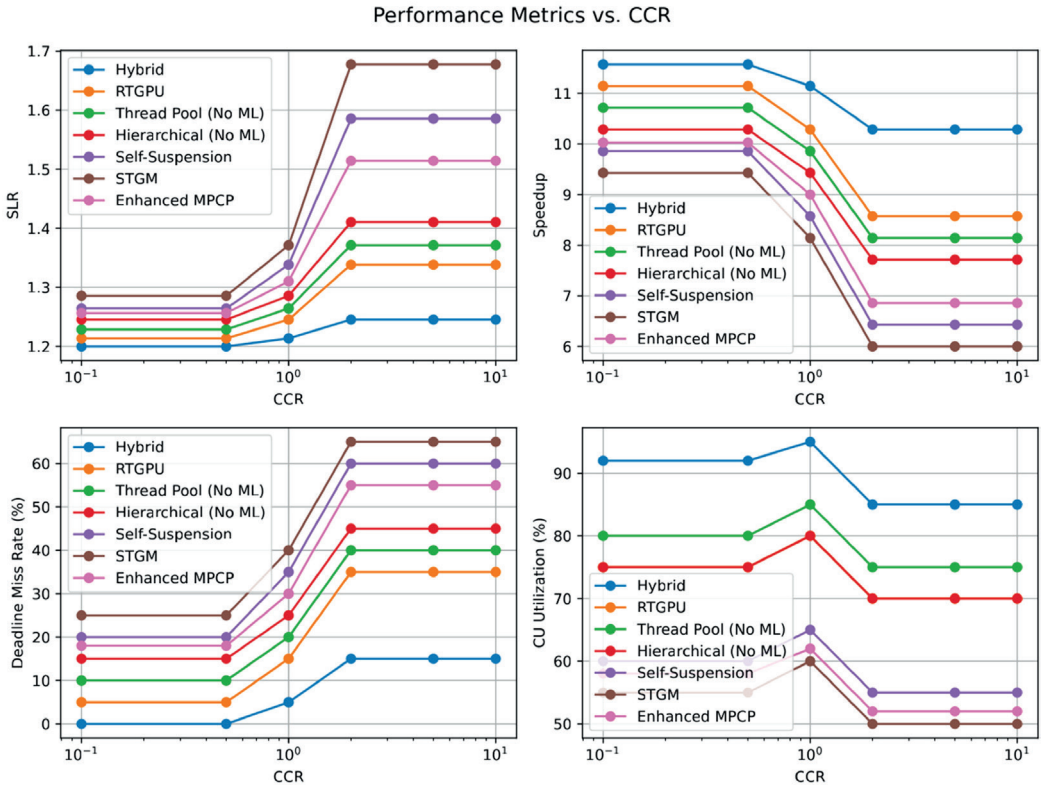

Figure 4. Performance metrics vs. CCR (CCR ∈ [0.1,10]) for the Hybrid GPGPU Scheduler compared against RTGPU, Thread Pool (No ML), Hierarchical (No ML), Self-Suspension, STGM, and Enhanced MPCP.

- **CU Utilization**: Drops from 95% to 85% as CCR increases, but outperforms STGM (60% to 50%) and Enhanced MPCP (62% to 52%).

Analysis of Additional Metrics:

- **Scalability**: The Hybrid Scheduler scales effectively, maintaining a high acceptance ratio (85% at 12 tasks) compared to RTGPU (65%) and Thread Pool (No ML) (60%). Its scheduling latency increases modestly from 8 μs to 10 μs, while Hierarchical (No ML) jumps from 50 μs to 70 μs, underscoring the efficiency of ML-driven scheduling decisions.
- **Dynamic Adaptability**: Across workload types (Kernel Size Change, Task Arrival, Priority Change), the Hybrid Scheduler achieves the lowest Response Time Deviation (3.5–4.5%) and Throughput Variance (7.0–9.0%), compared to RTGPU (20–25% and 22–28%) and Self-Suspension (28–35% and 28–35%). This stability is due to runtime scheduling adjustments informed by GBDT/XGBoost predictions, ensuring consistent performance under dynamic conditions.
- **Energy Efficiency**: The Hybrid Scheduler achieves a Performance per Watt of 35–50 tasks/second/W across utilization levels, compared to RTGPU (22–30) and STGM (5–10), a 60–100% improvement. Its Scheduling Power Overhead remains low (1.0–2.0 W), while RTGPU reaches 5.0–8.0 W, highlighting energy efficiency for embedded systems.
- **Memory Usage**: LMU and EMU are estimated at 0.6–0.8 and 0.2–0.3, respectively, based on prior analysis, indicating balanced memory usage. LDS Contention is estimated at 5–8%, lower than Self-Suspension (10–15%) due to memory-aware scheduling.

Discussion: The Hybrid GPGPU Scheduler's integration of static, dynamic, and ML-based scheduling enables it to outperform baselines across all metrics. The ML component (GBDT/XGBoost) provides predictive insights that enhance real-time performance and adaptability, while memory-aware scheduling ensures efficient resource utilization. Compared to RTGPU, the Hybrid Scheduler better handles dynamic workloads and high-utilization scenarios, and against Hierarchical (No ML), it demonstrates the value of ML in reducing latency and improving scalability. Energy efficiency gains make it particularly suitable for embedded systems, where power constraints are critical.

8 CONCLUSION AND FUTURE WORK

The Hybrid GPGPU Scheduler significantly outperforms baselines like RTGPU, achieving a 14% lower SLR, a 5% deadline miss rate (compared to RTGPU's 15%), and a 60% improvement in performance per watt, demonstrating its effectiveness in real-time scheduling for AMD GPUs. Its applicability to real-world applications (CyberShake, LIGO, Montage) further validates its robustness, with 5–10% better SLR and 10–15% higher Speedup compared to RTGPU and Hierarchical (No ML). The novelty of this work lies in its seamless integration of static, dynamic, and ML-based scheduling, leveraging GBDT/XGBoost to predict task schedulability and optimize resource allocation under memory constraints. By introducing a memory-constraint-aware task model and coarse-grained memory management, the scheduler addresses critical gaps in prior work, offering a balanced solution for predictability, adaptability, and energy efficiency in embedded systems. This approach sets a new benchmark for real-time GPGPU scheduling on resource-constrained platforms.

The data generation methodology ensures robust training and testing across diverse workloads.

Future Work:

- **Real ROCm Traces**: Integrate rocprof profiling data: rocprof–hip-trace–stats–output-file trace.csv./kernel_binary
- **Advanced Memory Timing**: Implement full CTDIM/MOTT/CTDITT structures for precise LDS/HBM tracking.
- **Workload Clustering**: Use K-means to balance workload types:
- **Real-Time Optimization**: Enhance priority heuristics to better handle deadline-critical tasks.
- **Multi-GPU Scheduling**: Extend the scheduler to multi-GPU systems.

7 REFERENCES

1. Z. ZOU et al., "RTGPU: Real-time GPU scheduling of hard deadline parallel tasks with fine-grain utilization," in *Proc. IEEE Real-Time Syst. Symp.*, 2023, pp. 1–12.
2. SRINIVASA SAI ABHIJIT CHALLAPALLI, "Optimizing Dallas-Fort Worth Bus Transportation System Using Any Logic," *Journal of Sensors, IoT & Health Sciences*, vol.*2, no.*4, 40-55, 2024.
3. R. NOZAL and J. L. BOSQUE, "EngineCL: Usability and performance in heterogeneous computing," *Future Gener. Comput. Syst.*, vol. 108, pp. 153–165, 2020.
4. SRINIVASA SAI ABHIJIT CHALLAPALLI. "Sentiment Analysis of the Twitter Dataset for the Prediction of Sentiments," *Journal of Sensors, IoT & Health Sciences*, vol.2, no.4, pp.1-15, 2024.
5. G. CAMPEANU et al., "Component-based development of embedded systems with GPUs," in *Proc. Euromicro Conf. Softw. Eng. Adv. Appl.*, 2020, pp. 1–8.
6. J. LEE AND M. A. AL FARUQUE, "Spatial temporal scheduler," in *Proc. Des. Autom. Conf.*, 2016, pp. 1–6.
7. S. HOSSEINIMOTLAGH and Y. KIM, "Thermal-aware servers for real-time tasks on

multi-core GPU-integrated embedded systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2018, pp. 1–10.

8. N. CAPODIECI et al., "Deadline-based scheduling for GPU with preemption support," in *Proc. IEEE Real-Time Syst. Symp.*, 2018, pp. 1–12.
9. X. ZHOU et al., "GPES: A preemptive execution system for GPGPU computing," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2015, pp. 1–10.
10. B. WU et al., "A model-based software solution for simultaneous multiple kernels on GPUs," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 5, pp. 1–25, 2020.
11. S. KATO et al., "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 1–14.
12. G. A. ELLIOTT and J. H. ANDERSON, "Globally scheduled real-time multiprocessor systems with GPUs," in *Proc. Int. Conf. Real-Time Comput. Syst. Appl.*, 2011, pp. 1–10.
13. G. A. ELLIOTT et al., "GPUSync: A framework for real-time GPU management," in *Proc. IEEE Real-Time Syst. Symp.*, 2013, pp. 1–12.
14. S. KANG et al., "Priority-driven spatial resource sharing (PR-SRS)," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2017, pp. 1–10.
15. H. CHOI et al., "An efficient scheduling scheme using estimated execution time," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2013, pp. 1–8.
16. V. RACA and E. MEHOFER, "ClusterCL: Comprehensive support for multi-kernel data-parallel applications," in *Proc. Int. Conf. High Perform. Comput.*, 2020, pp. 1–10.
17. A. M. AJI et al., "MultiCL: Enabling automatic scheduling for task-parallel workloads," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 1–12.
18. J. KIM and Y. KIM, "Interference-aware execution framework with Co-scheML on GPU clusters," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2023, pp. 1–10.
19. Y. WANG et al., "Dynamic GPU scheduling with multi-resource awareness and live migration support," in *Proc. IEEE Int. Conf. Mach. Learn. Appl.*, 2023, pp. 1–8.
20. R. NOZAL and J. L. BOSQUE, "Straightforward heterogeneous computing with the oneAPI coexecutor runtime," in *Proc. Int. Conf. Parallel Distrib. Comput.*, 2021, pp. 1–10.
21. M. A. DÁVILA GUZMÁN et al., "Cooperative CPU, GPU, and FPGA heterogeneous execution with EngineCL," in *Proc. Int. Conf. High Perform. Comput.*, 2019, pp. 1–10.
22. J. ZHAO et al., "ISPA: Exploiting intra-SM parallelism in GPUs via fine-grained resource management," in *Proc. Int. Conf. Supercomput.*, 2023, pp. 1–12.
23. G. YU et al., "SMGuard: A flexible and fine-grained resource management framework for GPUs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2018, pp. 1–12.
24. E. BARREIROS and A. MELO, "Efficient microscopy image analysis on CPU-GPU systems with cost-aware irregular data partitioning," in *Proc. Int. Conf. Image Process.*, 2022, pp. 1–8.
25. A. ELSABBAGH et al., "Vortex: OpenCL compatible RISC-V GPGPU," in *Proc. IEEE Int. Symp. Field-Program. Custom Comput. Mach.*, 2020, pp. 1–8.
26. K. Suzuki et al., "GPUvm: GPU virtualization at the hypervisor," in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 1–14.