WAMP^{2}S: Workload-Aware GPU Performance Model Based Pseudo-Preemptive Real-Time Scheduling for the Airborne Embedded System

Yuan Yao, *Member, IEEE*, Shuangyang Liu, Sikai Wu, Jinyu Wang, Jinting Ni, Gang Yang, *Member, IEEE*, and Yu Zhang, *Member, IEEE*

Abstract—New generation airborne embedded system has deployed Graphical Processing Units (GPUs) to raise processing capability to meet growing computational demands. Comparing with the cloud system, the airborne embedded system usually has a fixed application set, but strict real-time constraints. Unfortunately, the inherent GPU scheduler does not consider the application priority, which cannot provide the sufficient real-time capability to the airborne embedded system. To meet timeliness requirements, it is necessary to predict timing behaviors of those applications and design a real-time scheduling policy based on priority and deadline. We therefore propose WAMP^{2}S, a workload-aware GPU performance model based pseudo-preemptive real-time scheduling algorithm for the airborne embedded system. The workload-aware GPU performance model can accurately predict the execution time of an application, which is running concurrently with other applications on GPU. The pseudo-preemptive real-time scheduling algorithm can provide the approximate preemption by dynamically adjusting GPU computing resources for active applications. Unlike previous work on GPU performance model and GPU real-time scheduling, WAMP^{2}S considers the impact of co-executing workload on the execution time estimation and provides a software-only approach for preemption support. In addition, WAMP^{2}S implements a prototype GPU scheduler without any source code analysis. We evaluate the proposed GPU performance model and real-time scheduling algorithm in both simulated and realistic application sets. Experimental results illustrate that WAMP^{2}S can achieve low prediction error and high scheduling success ratio.

Index Terms—Real-time scheduling, GPU performance model, embedded system, preemptive scheduling

1 Introduction

With the improvement of intelligence level of Unmanned Aerial Vehicle (UAV), more and more complex tasks are deployed on it. This makes a strong demand of computational capacity for airborne embedded system to deal with an increasing number of applications such as mission allocation, path planning, radar signal processing, object detection, target tracking and so on. In the past few years, Graphics processing units (GPUs) become major computing resources of the cloud and server infrastructures for scientific computing tasks such as image processing [1], [2], data mining [3], [4], biomedical simulating [5], [6] and financial analyzing [7], [8]. With the widespread use of GPU in diverse industrial fields, most of advanced airborne embedded systems are also deployed GPUs as accelerators to improve performance.

• *The authors are with the School of Computer Science, Northwestern Polytechnical University, Xi'an 710072, China. E-mail: {yaoyuan, zhangyu}@nwpu.edu.cn, {liushuangyang, wusikai, jinyuwang, nijinting}@mail.nwpu.edu.cn, yang.gang@gmail.com.*

Manuscript received 26 Aug. 2021; revised 3 Nov. 2021; accepted 5 Dec. 2021. Date of publication 13 Dec. 2021; date of current version 23 May 2022.
This work was supported in part by the National Natural Science Foundation of China under Grants 61876151 and 62032018, in part by the Shanghai Pujiang Program under Grant 19PJ1430900, and in part by the Fundamental Research Funds for the Central Universities under Grant 3102019DX1005. (Corresponding author: Yuan Yao.)
Recommended for acceptance by A.J. Peña, M. Si and J. Zhai.
Digital Object Identifier no. 10.1109/TPDS.2021.3134269

However, there is a big difference between the cloud platform and the airborne embedded system. Usually, the scheduler of the cloud platform is aim to achieve high efficiency and throughput. Nevertheless, the airborne embedded system is a typical real-time system which has strict timing constraints for applications. The scheduling goal of such a system is to guarantee timeliness. Therefore, to accurately predict timing behavior of an executing application is crucial for such systems. To handle this problem, researchers have established various analytical GPU performance models to estimate the application execution time [9], [10], [11], [12], [13], [14], [15], [16]. However, these models only predict the performance considering the situation that a single application runs on the GPU.

The co-execution of multiple applications on a single GPU was first supported by NVIDIA Fermi architecture. Then, NVIDIA's Kepler GPUs adopt Hyper-Q technology that employ 32 hardware queues to avoid false dependencies between computations [18]. They can achieve higher efficiency of concurrent execution than Fermi GPUs. When multiple applications concurrently run on a single GPU, each application suffers significant execution performance degradation caused by the resource contention. To motivate our work, we conduct an experiment to measure the execution time of an application lavaMD which runs concurrently with different number of other applications. As shown in Fig. 1, we observe that execution time of lavaMD is greatly increasing with the growth of co-executing applications. It is challenging to establish an accurate performance model

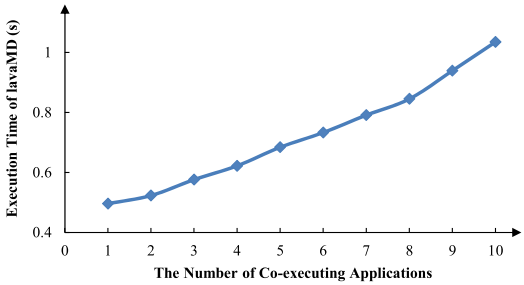


Fig. 1. The impact of co-execution on GPU performance.

to predict the execution time of co-executing applications since there are various resources (CUDA core, global memory, shared memory, cache) involved in concurrent execution.

The other challenging issue is the implementation of real-time scheduling on GPU. Once CUDA kernels are running on the GPU device, these kernels are dispatched by hardware scheduler. Since the hardware scheduler dose not take the priority into account, the real-time requirement of kernels with high priority cannot be guaranteed. Moreover, due to the asynchronous and non-preemptive nature of GPU, a high priority application may be blocked by a low priority application. This priority inversion makes it hard to ensure the real-time schedulability. Previous studies have investigated GPU real-time scheduling problem and proposed some preemptive supporting methods [20], [21], [22], [23], [27], [28], [29], [30], [32]. However, these methods either need to modify the device driver or insert scheduling point into source code.

In this paper, we propose, WAMP^{2}S, a real-time scheduling method for the airborne embedded system based on workload-aware GPU performance model. For a specific application App_i, we define workload as the other applications concurrently executing with App_i on a single GPU.

The major contributions of this paper are four-fold:

1) We propose a workload-aware GPU performance model that can accurately predict the execution time of an application when it is running concurrently with other applications on the GPU device;
2) We adopt a lightweight Self-Organizing Fuzzy Neural Network (SOFNN) to implement the GPU performance model without source code analysis which can greatly reduce the scheduling overhead;
3) We design a pseudo-preemptive scheduler based on GaiaGPU [34], a GPU resource sharing framework, which can dynamically adjust computing resources for co-executing applications. When a high-priority application arriving, the scheduler can provide the approximate preemption by decreasing the computing resources of the low-priority application. Moreover, the pseudo-preemptive scheduling algorithm also does not require any source code analysis;
4) We implement a software prototype on a the real airborne embedded system. We evaluate the proposed GPU performance model and real-time scheduling algorithm in both simulated and realistic application sets. The results prove that WAMP^{2}S can meet the soft real-time deadline requirement.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces the overall system architecture. Section 4 describes the workload-aware GPU performance model. Section 5 presents the pseudo-preemptive real-time scheduling algorithm. Section 6 validates the proposed model and algorithm on the real airborne embedded system with both simulated and realistic application sets. Finally, Section 7 draws the conclusion.

2 Related Work

2.1 GPU Performance Model

GPU performance model is aiming to predict the execution time of applications running on the GPU platform by considering various characteristics, such as the number of cores, memory latency, program abstraction and so on. Nath *et al.* propose the pipeline models to reveal the relationship between computation cycles and memory latency [9]. Wu *et al.* use K-means clustering that builds a performance model of a kernel at a single configuration to predict performance across other configurations [10]. Dao *et al.* use a machine learning based model to improve the proposed linear performance model [11]. Wang *et al.* propose a hybird framework that combines source code analysis and trace-based simulation to predict the execution time of CUDA kernels [12]. Lew *et al.* adopt simulation method that combines performance counters and specific hardware information to estimate the kernel execution time [13]. Lym *et al.* design a GPU performance model, DeLTA, specially for the deep learning applications like CNNs. DeLTA accurately models traffic across all memory hierarchy levels [14]. Wu *et al.* take resources and data transfer cost between CPU and GPU into account to predict massively parallel programs execution time [15]. Wang *et al.* establish a novel GPU pipeline performance model with GPU memory model to predict kernel execution time under different core and memory frequencies [16]. However, some works need source code analysis to get more program configurations, which are inapplicable to those closed source applications. In addition, all these models only predict a single application runs on the GPU without co-executing workload. Wang *et al.* improve their previous performance model [12] to estimate the kernel execution time with parallel OpenCL workloads on GPUs [17]. But it still needs to transform the kernel source code into LLVM intermediate representation instructions. So far, few works study the GPU performance model for multiple applications co-executing on the GPU platform without source code analysis.

2.2 GPU Real-Time Scheduling

The main purpose of conventional GPU is to accelerate high-performance computing applications. However, it lacks a key technology for real-time systems, i.e., preemptive scheduling. If a low-priority application requests the GPU, the inherent scheduler will process this request in an efficient manner. Usually, the computational resources are allocated to this application as many as possible. When a high-priority application comes, it may be blocked or delayed by the low-priority application. This is known as priority inversion [19] since the default GPU scheduling policy has no regard for the application priority. With the widely use of GPU in

embedded systems with many highly-critical applications such as advanced driver assistance systems and airborne embedded systems, it is necessary to study real-time scheduling technology for GPUs. Kato *et al.* propose TimeGraph [20], RGEM [21] and Gdev [22]. TimeGraph is a non-preemptive GPU scheduler specifically for graphic applications. It provides fixed-priority scheduling and resource reservation capabilities at the device driver level. RGEM provides preemptive scheduling of memory-copy transactions by splitting a non-preemptive DMA operation into multiple smaller chunks. Gdev provides first-class GPU resource management schemes for multi-tasking systems. It is implemented as a kernel-space open-source GPU driver. Elliott *et al.* propose GPUsync [23], a real-time GPU management framework which views GPU scheduling as a synchronization problem. It provides a broad range of methods such as kernel migration, budget enforcement and scheduling policy configuration. However, these methods are based on an open-source GPU device driver. Thus, the implementation of these methods needs to modify OS kernel and device driver.

Some studies implement hardware-based preemptive scheduling technology. Tanasic *et al.* propose hardware extensions to enable the preemption support [24]. It saves all contexts of intercepted thread blocks and reloads these contexts when resuming execution. Park *et al.* propose a preemption method called flushing [25]. It drops contexts of preempted thread blocks immediately. Wang *et al.* implement the spatial preemption for GPU [26]. Instead of removing contexts of a whole grid, it only intercepts thread blocks as little as possible. However, the hardware-based preemption suffers significant context-switching overhead.

Software-based approaches are investigated to support kernel preemption, which is so called kernel slicing technology. Zhong *et al.* present Kernelet [27] that divides a large kernel into small sub-kernels. The preemption can be executed at the end of each sub-kernel. Zhou *et al.* implement GPES [28] that performs both kernel slicing and data slicing. GPES develops a source-to-source transformation technique, which can compile the transformed kernel code into CUDA binaries. Wu *et al.* propose a flexible kernel preemption FLEP supporting both temporal and spatial preemptions [30]. Instead of kernel level, FLEP is performed at the block-task granularity level. Chen *et al.* propose an efficient preemptive scheduling framework Effisha [29]. Effisha also works at the block-task level. It requires no kernel slicing, but automatically inserts a scheduling point at the end of every block. Hartmann *et al.* put forward GPUart [31], a light-weight high-level approach with limited preemption. GPUart inserts fixed scheduling points into the source code. The scheduler checks whether there is a high-priority application comes at each scheduling point. Lee *et al.* propose an idempotence-based preemptive GPU scheduling for embedded system [32]. The kernel idempotence classifier analyzes the source code to determine kernel type. The kernel transactionization scheme creates a snapshot of the GPU memory before executing non-idempotent kernels. The memory snapshot will be reloaded when the preempted kernel is resumed. Despite the software-based preemption has less context-switching overhead compared with hardware-based technology, to determine the granularity of the sub-kernels is still a challenging issue. Moreover, either the kernel slicing methods or scheduling point inserting schemes need to analyze and compile the source code, which is not applicable for those closed source systems.

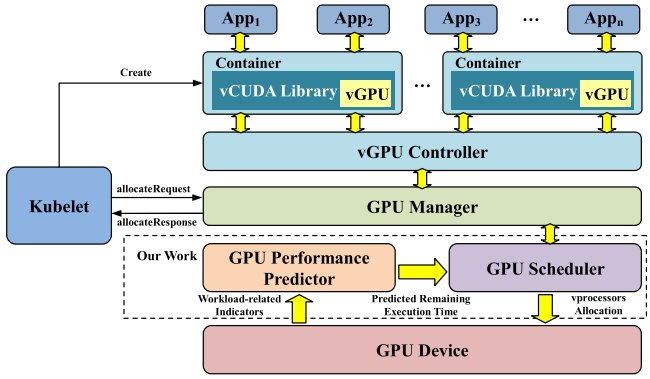


Fig. 2. System architecture.

3 System Architecture

The proposed WAMP²S scheduler for the airborne embedded system is based on GaiaGPU [34], a GPU resource sharing framework. The system architecture of WAMP²S is shown in Fig. 2. The whole system is composed of six parts.

1) Kubelet. Kubernetes provides the device plugin framework to advertise computing resources such as GPUs and FPGAs. A device plugin is registered on the Kubelet via gRPC service [35]. Then, the Kubelet could get device information from the device plugin and take in charge of device resources.

2) vCUDA Library. GaiaGPU adopts vCUDA [33] to implment GPU virtualization. The vCUDA Library is a wapper library that intercepts and redirects CUDA calls from the application. vGPU is created and used by the vCUDA Library. vGPU abstracts several features of the real GPU to give each application a complete view of the underlying hardware. vGPU also creates a virtual GPU context for each CUDA application. In GaiaGPU, the vCUDA Library is running in the container (i.e., Nvidia-docker).

3) vGPU Controller. vGPU Controller maintains a list of containers assigned with vGPUs. It monitors the status of the container. If a container is dead, vGPU Controller removes the container from the list.

4) GPU Manager. GPU Manager is an instantiated device plugin to advertise GPU resources to the Kubelet. When the GPU Manager is successfully registered on the Kubelet, it returns the device list to the Kubelet. Instead of the physical GPU, GPU Manager sends the vGPU information to the Kubelet. vGPU is virtualized in two resource dimensions: vmemory and vprocessor. In GaiaGPU, a vmemory device is a memory unit with 256MB storage and a vprocessor device is 1% of GPU CUDA cores (a physical GPU has 100 vprocessor devices). When an application launch a CUDA request in a container, the Kubelet will send an *allocateRequest* to the GPU Manager. Then, the GPU Manager sends a scheduling request to the GPU scheduler. If the application is successfully dispatched to the physical device, the GPU Manager returns an *allocateResponse* with configurations for accessing the allocated devices to the Kubelet.

5) GPU Performance Predictor. We add the GPU Performance Predictor based on the proposed workload-aware

TABLE 1
The Workload-Related Indicators and CUPTI Events

Resources	Indicators	Events
Stream Multi-processor	*instructions*	inst_executed
	warps	active_warps
L2 Cache	*l2_read*	l2_subp0_total_read_sector_queries+l2_subp1_total_read_sector_queries
	l2_write	l2_subp0_total_write_sector_queries+l2_subp1_total_write_sector_queries
Global Memory	*global_load*	global_load
	global_store	global_store
Shared Memory	*shared_load*	shared_ld_transactions
	shared_store	shared_st_transactions
	shared_ldbconf	shared_ld_bank_conflict
	shared_stbconf	shared_st_bank_conflict
Texture Memory	*texture_load*	tex0_cache_sector_queries+tex1_cache_sector_queries

GPU performance model into GaiaGPU framework. It predicts the execution time of co-executing applications according to some workload-related indicators, which are measured by CUDA Profiling Tool Interface (CUPTI) [36].

6) GPU Scheduler. GPU scheduler receives scheduling requests from the GPU Manager and allocates resources for applications. We redesign the GPU scheduler in GaiaGPU and implement a pseudo-preemptive scheduling policy. It dynamically adjusts vprocessor resources for active applications to meet the real-time requirement rather than removes the low-priority application at the scheduling point.

Comparing with GaiaGPU framework, the main improvements of our work is summarized as follows:

- GaiaGPU framework only implements a static computing resource allocation scheme. The portion of vprocessors assigned to a container is saved in the configuration file, which cannot be changed at runtime. We provide a runtime API that can reallocate vprocessors to the container dynamically;
- We add a new component, i.e., the GPU Performance Predictor, to the framework. It implements the proposed workload-aware GPU performance model which can estimate the execution time of co-executing GPU applications;
- We design a pseudo-preemptive scheduling policy in the GPU scheduler. It supports approximate preemption and soft real-time deadlines without any source code analysis.

4 Workload-Aware GPU Performance Model

In this section, we introduce the proposed Workload-Aware GPU Performance Model in detail [37]. First, some GPU workload-related indicators are described. Then, a Self-Organizing Fuzzy Neural Network (SOFNN) is put forward to predict the GPU performance. Finally, we show how to train the SOFNN with a given application set.

4.1 GPU Workload-Related Indicators

Co-executing applications on a single GPU lead to resource contention which greatly degrades GPU performance. For a specific application App_i, we define the workload as the other applications concurrently executing with application App_i. Thus, the GPU performance model needs to collect the workload-related indicators as inputs. In this paper, we consider 5 major GPU resources including stream multi-processor, L2 cache, global memory, shared memory and texture memory. The workload-related indicators are detailed in Table 1.

An event listed in Table 1 is a countable activity, action, or occurrence on GPU device. CUPTI provides Event API to sample these event values for CUDA applications at runtime. A selected GPU resource includes one or more workload-related indicators. The detailed description is given below.

1) Stream Multi-Processor. The Stream Multi-processor (SM) event values typically represent activity or action of thread warps. We chose three indicators (*instructions*, and *warps*) that correspond to three CUPTI events (inst_executed and active_warps). The inst_executed is the number of instructions executed. The active_warps is the number of active warps.

2) L2 Cache. CUPTI Event API provides several hardware counters to address the performance of the L2 cache. We focus on two major activities: L2 cache read and write (*l2_read*, *l2_write*). L2 cache read is calculated by adding values of l2_subp0_total_read_sector_queries and l2_subp1_total_read_sector_queries. Each event represents the number of read requests to a subpartition of the L2 cache which increments by 1 for each 32-byte access. The calculation of L2 cache write is the similar to L2 cache write.

3) Global Memory. The global memory is off-chip memory (DRAM on the GPU) which has a high access latency. Two indicators (*global_load*, *global_store*) are chosen to represent the global memory access intensity. The corresponding events are global_load and global_store. The global_load is the number of global memory load transactions from SM. The global_store is the number of global memory store transactions from SM.

4) Shared Memory. The shared memory is on-chip memory which is roughly 100x faster than the global memory if there are no shared bank conflicts between the threads. However, a large number of bank conflicts may cause that the shared memory access latency is higher than global

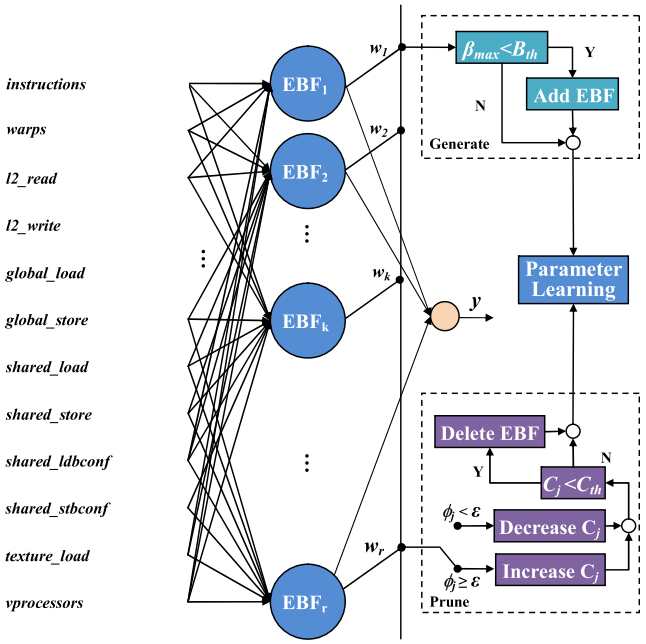


Fig. 3. The structure of SOFNN.

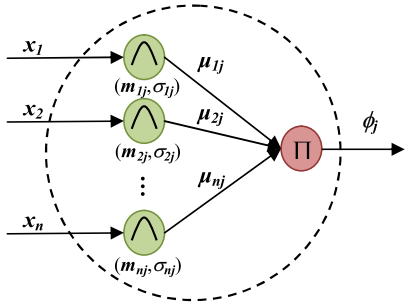


Fig. 4. The j^{th} EBF unit.

memory latency. Therefore, two indicators (*shared_load*, *shared_store*) are selected to describe the shared memory access intensity. The other two indicators (*shared_ldbconf*, *shared_stbconf*) are chosen to illustrate the shared bank conflicts occurred during runtime. Correspondingly, shared_load_transactions, shared_store_transactions, shared_ld_bank_conflict and shared_st_bank_conflict are the number of shared memory load transactions, shared memory store transactions, shared bank conflicts due to load requests and due to store requests, respectively.

5) Texture Memory. The texture memory is a variety of read-only memory. The memory space is cached on chip. Although texture memory is originally designed for traditional graphics applications, it can also be used quite effectively in GPU computing applications. The indicator *texture_load* is calculate by adding values of tex0_cache_sector_queries and tex1_cache_sector_queries. Each event represents the total number of load requests to a subpartition of the texture cache which increments by 1 for each 32-byte access.

There is another important input variable of the SOFNN, i.e., the number of *vprocessors* (detailed in Section 5) allocated to an application, which represents the usage of computing resources for this application. It is not obtained by CUPTI, but is set dynamically by the GPU scheduler.

4.2 SOFNN Based GPU Performance Model

We adopt a Self-Organizing Fuzzy Neural Network (SOFNN) [38] to model the GPU performance with different workloads. SOFNN can efficiently model non-linear dependencies among different inputs based on fuzzy inference. In addition, it also has the learning ability of neural network to optimize the network structures and parameters by training.

The structure of SOFNN for GPU performance estimation is shown in Fig. 3. It is a four-layer network. The first layer is the input layer which contains all workload-related indicators. The second and third layers can be seen as multiple Ellipsoidal Basis Function (EBF) units. As shown in Fig. 4, an EBF unit includes the membership function layer and the fuzzy rule layer. Each EBF unit is a premise part (IF-part) of the fuzzy rule. Nodes in the membership function layer represent the fuzzy linguistic level of the input variables. For an input x_i, the Gaussian membership function is used

$$\mu_{ij} = exp\left[-\frac{(x_i - m_{ij})^2}{2\sigma_{ij}^2}\right], i = 1, 2 \ldots n, j = 1, 2 \ldots R, \quad (1)$$

where μ_{ij}, m_{ij} and σ_{ij} are the output, mean and the deviation of the Gaussian membership function of the i^{th} input x_i in the j^{th} EBF unit. R is the number of generated fuzzy rules. The fuzzy rule layer uses the product operation as the fuzzy AND operation to determine the firing strength of each rule. The firing strength ϕ_j of the j^{th} rule is as follows:

$$\phi_j = \prod_{i=1}^{n} \mu_{ij}, j = 1, 2, \ldots, R. \quad (2)$$

Nodes in output layer perform defuzzification. The outputs of the network are given as follows:

$$y = \frac{\sum_{j=1}^{R} \phi_j w_j}{\sum_{j=1}^{R} \phi_j}, \quad (3)$$

where w_j is the connect weight between the jth rule and the output as a consequent part (THEN-part) of the fuzzy rule. Finally, the output y is the predicted execution time of the application.

4.3 SOFNN Learning Algorithm and Training Data

4.3.1 SOFNN Learning Algorithm

As shown in Fig. 3, SOFNN has two learning phase: structure learning and parameter learning. The structure learning is to generate or prune EBF units of SOFNN. From Fig. 4, an EBF unit can be represented by three vectors

$$\begin{aligned} M_j &= [m_{1j}, m_{2j}, \ldots, m_{nj}]^T \\ \Sigma_j &= [\sigma_{1j}, \sigma_{2j}, \ldots, \sigma_{nj}]^T \\ W_j &= [w_{1j}, w_{2j}, \ldots, w_{mj}]^T. \end{aligned} \quad (4)$$

The firing strength ϕ_j of the j^{th} EBF unit indicates the degree that the input vector X^k belongs to this unit. $X^k = \{x_1^k, x_2^k, \ldots, x_n^k\}$ is the k^{th} sample of input variables. Define the maximum firing strength β_{max} as follows:

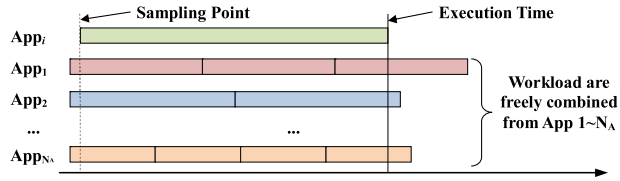


Fig. 5. SOFNN training data sampling.

$$\beta_{\max} = \max_{1 \le j \le R_k} \phi_j, \tag{5}$$

where R_k is the number of the EBF units when the k^{th} input sample is entering. If $\beta_{max} \le B_{th}$ ($B_{th} \in (0,1)$ is a predefined threshold), it indicates that the input sample is on the edge of all the existing membership functions. Thus, a new EBF unit will be generated to contain this input sample which is formed by three vectors as below:

$$\begin{aligned} M_{new} &= X^k \\ \Sigma_{new} &= \Sigma_0 \\ W_{new} &= U(-1,1), \end{aligned} \tag{6}$$

where Σ_0 is a constant vector that all elements are initialized as 0.01. Function $U(a,b)$ generates uniformly distrusted random numbers range from a to b.

The pruning operation is to remove those unimportant EBF units. We define the critical degree (C_j) of the j^{th} EBF unit which represents the correlation between the input and the j^{th} fuzzy rule. It is calculated as follows:

$$C_j = \begin{cases} \frac{C_j}{1+\exp(-\alpha_1 C_j)} & \phi_j < \varepsilon \\ C_j + \frac{C_j}{1+\exp(\alpha_2 C_j)} & \phi_j \ge \varepsilon \end{cases}, \tag{7}$$

where α_1 and α_2 are designed constants. If the firing strength ϕ_j of the j^{th} EBF unit is smaller than ε (a predefined threshold), it indicates that the j^{th} fuzzy rule and the input variables has a weak correlativity. Then, the critical degree of j^{th} rule will be decreased. Otherwise, the critical degree needs to be increased if ϕ_j is greater than ε. If the critical degree C_j drops down to a lower bound C_{th}, the j^{th} EBF unit should be deleted.

The parameter learning adopts the linear least square and supervised gradient descent. Due to space limitations, we skip the introduction of parameter learning phase. The details of this part can refer to our previous work [38].

4.3.2 SOFNN Training Data

Usually, the airborne embedded system has a fixed application set, applications running on GPU are deterministic. Therefore, the SOFNN is trained based on a fixed application set (Γ) with N_A applications. We implement a monitor thread to collect all CUPTI events of selected workload-related indicators as input variables for SOFNN. In order to train an SOFNN for a specific application App_i under different workload circumstances and computing resources, applications in Γ except App_i form different combinations and are co-executing with App_i allocated by different vprocessors. The sampling of input and output variables of SOFNN is shown in Fig. 5.

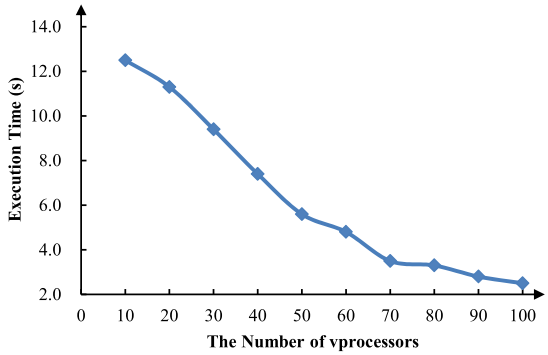


Fig. 6. The impact of vprocessors on execution time.

Each application runs continuously until App_i is finished. This is to ensure that the workload is stable during the execution of App_i. The monitor thread samples input vector X^k at the beginning when App_i starts. As App_i is finished, the execution time of App_i can be recorded as the output y^k. Then, an input-output pair $\langle X^k, y^k \rangle$ of a workload combination at a given number of vprocessors is obtained. Finally, we can train SOFNN when getting enough samples from different workload combinations at various vprocessor allocations.

5 Pseudo-Preemptive Real-Time Scheduling

In this section, the proposed pseudo-preemptive real-time scheduling algorithm is introduced in detail. First, we give a GPU application model and show how to calculate the remaining time of an application when workload is changed. Then, we describe the scheduling policy based on greedy algorithm. Finally, we discuss some optimizations on the basic implementation to further reduce the scheduling overhead.

5.1 Deadline-Based Pseudo-Preemptive Scheduling

The inherent GPU scheduler does not take the application priority into account. It is efficient for Best-Effort applications, but cannot sufficiently support applications with tight real-time requirements in the airborne embedded system. Many real-time scheduling methods are proposed to solve this problem, however, most of them need to insert scheduling points and recompile the source code. In addition, frequent context-switching operations of the preemptive scheduling will cause a high scheduling overhead. Therefore, we propose a pseudo-preemptive scheduling approach based on GaiaGPU. GaiaGPU implements a device plugin of Kubernetes that can provide virtualized computing resources (i.e., vprocessor devices) to a GPU application with 1% granularity. Thus, a physical GPU can be seen as 100 virtualized vprocessors. Fig. 6 shows that an application is running on GPU with different vprocessor allocations. Obviously, the more the vprocessor is allocated, the faster the application runs.

Unlike previous preemptive GPU scheduling methods that remove a sub-kernel of an application from GPU device, the proposed pseudo-preemptive scheduling dynamically adjusts vprocessor allocations for active applications according to the priority and deadline. Consider an extreme case that a high-priority application is co-executing with a low-priority application. If there is only one vprocessor

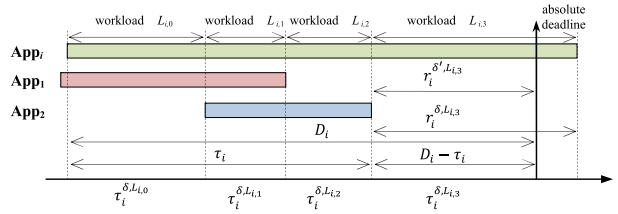


Fig. 7. The remaining time prediction of App_i.

allocated to the low-priority application, the high-priority application is approximate to use the whole GPU computing resources.

We consider an application model with constrained deadlines. Specifically, a GPU application is characterized as follows:

$$App_i = \left\langle C_i^{\delta,N}, C_i^{\delta,L}, \tau_i^{\delta,L}, r_i^{\delta,L}, D_i, T_i, P_i \right\rangle. \tag{8}$$

- $C_i^{\delta,N}$ is the measured worst-case execution time (WCET) of App_i when it runs individually without any workload on δ vprocessors ($\delta \in [1, 100]$). Thus, for a given set Γ with N_A applications, we maintain a $N_A \times 100$ matrix (M_{WCET}^N) to store WCET of all applications on different computing resources;
- $C_i^{\delta,L}$ is the predicted execution time of App_i when it runs concurrently with the stable workload L on δ vprocessors;
- $\tau_i^{\delta,L}$ is the actual execution time that App_i is co-executing with workload L on δ vprocessors;
- $r_i^{\delta,L}$ is the predicted remaining execution time of App_i with workload L;
- D_i is the relative deadline;
- T_i is the period for periodic applications or the minimum inter-arrival time for sporadic applications;
- P_i is the fixed priority of App_i.

In order to meet the real-time requirement of a given application set, the key point is to guarantee the remaining execution time is less than the relative deadline minus the actual execution time for all applications.

Once the App_i launching CUDA request, the GPU performance predictor will estimate the execution time $C_i^{\delta,L}$ of App_i based on the proposed SOFNN model with workload-related indicators and vprocessors. Thus, the remaining time $r_i^{\delta,L}$ is equal to $C_i^{\delta,L}$ if the workload L is stable during the execution of App_i. However, the workload usually changes when an existing application is finished or a new application is launched. Here, we take an example shown in Fig. 7 to illustrate how to estimate the remaining execution time when the workload are changed.

Let J_i be change times of workload and $L_{i,j}$ be the j^{th} workload during the execution of App_i. When the workload is changed, the GPU performance predictor will recalculate the remaining execution time of App_i.

First, it makes a linear mapping between the actual execution time $\tau_i^{\delta,L_{i,0}}$ of App_i with workload $L_{i,0}$ and the actual execution time $\tau_i^{\delta,N}$ of App_i with no-load

$$\tau_i^{\delta,N} = \frac{C_i^{\delta,N}}{C_i^{\delta,L_{i,0}}} \tau_i^{\delta,L_{i,0}}. \tag{9}$$

Then, the remaining time $r_i^{\delta,L_{i,1}}$ of the first workload change is calculated according to $\tau_i^{\delta,N}$, $C_i^{\delta,N}$ and $C_i^{\delta,L_{i,1}}$

$$\begin{aligned} r_i^{\delta,L_{i,1}} &= \frac{C_i^{\delta,L_{i,1}}}{C_i^{\delta,N}} \left(C_i^{\delta,N} - \tau_i^{\delta,N} \right) \\ &= \frac{C_i^{\delta,L_1}}{C_i^{\delta,N}} \left(C_i^{\delta,N} - \frac{C_i^{\delta,N}}{C_i^{\delta,L_{i,0}}} \tau_i^{\delta,L_{i,0}} \right) \\ &= C_i^{\delta,L_{i,1}} - \frac{C_i^{\delta,L_{i,1}}}{C_i^{\delta,L_{i,0}}} \tau_i^{\delta,L_{i,0}} \\ &= C_i^{\delta,L_{i,1}} \left(1 - \frac{\tau_i^{\delta,L_{i,0}}}{C_i^{\delta,L_{i,0}}} \right). \end{aligned} \tag{10}$$

Similarly, we can obtain the rest remaining time when workload is changed. For the j^{th} change of workload, we give a general expression of the remaining execution time of App_i as follows:

$$r_i^{\delta,L_{i,j}} = \begin{cases} C_i^{\delta,L_{i,0}} & j = 0 \\ C_i^{\delta,L_{i,j}} \left(1 - \sum_{k=0}^{j-1} \frac{\tau_i^{\delta,L_{i,k}}}{C_i^{\delta,L_{i,k}}} \right) & j > 0 \end{cases}. \tag{11}$$

If the predicted remaining execution time $r_i^{\delta,L_{i,j}}$ is greater than the relative deadline D_i minus the actual execution time τ_i, it is necessary to adjust the vprocessor resources for App_i. Assume that the number of vprocessor is increased from δ to δ', we have

$$\begin{aligned} \frac{r_i^{\delta',L_{i,j}}}{r_i^{\delta,L_{i,j}}} &= \frac{C_i^{\delta',L_{i,j}}}{C_i^{\delta,L_{i,j}}} = \frac{C_i^{\delta',N}}{C_i^{\delta,N}} \\ r_i^{\delta',L_{i,j}} &= \frac{C_i^{\delta',N}}{C_i^{\delta,N}} r_i^{\delta,L_{i,j}}. \end{aligned} \tag{12}$$

The adjustment of vprocessors should ensure that remaining execution time $r_i^{\delta',L_{i,j}}$ is less than the relative deadline D_i minus the actual execution time τ_i, so we have

$$\begin{aligned} r_i^{\delta',L_{i,j}} &\le D_i - \sum_{k=0}^{j-1} \tau_i^{\delta,L_{i,k}} \\ \frac{C_i^{\delta',N}}{C_i^{\delta,N}} r_i^{\delta,L_{i,j}} &\le D_i - \sum_{k=0}^{j-1} \tau_i^{\delta,L_{i,k}} \\ C_i^{\delta',N} &\le \left(D_i - \sum_{k=0}^{j-1} \tau_i^{\delta,L_{i,k}} \right) \frac{C_i^{\delta,N}}{r_i^{\delta,L_{i,j}}}. \end{aligned} \tag{13}$$

In Eq. (13), D_i is the relative deadline of App_i, the actual execution time $\tau_i^{\delta,L_{i,j}}$ is recorded when the j^{th} workload changes, the remaining execution time $r_i^{\delta,L_{i,j}}$ is predicted in Eq. (11), and $C_i^{\delta,N}$ can be obtained from the matrix M_{WCET}^N. Finally, we can find the minimum δ' of App_i in M_{WCET}^N to satisfy the above inequation.

5.2 Priority-Based Greedy Scheduling Policy

According to above deduction, we implement the pseudo-preemptive real-time scheduling policy based on the greedy algorithm, which is given in Algorithm 1. All

$App_i \in \Gamma$ are sorted in descending order according to the application priority. The larger the value of P_i, the higher the application priority ($P_i > P_{i+1}, i = \{1, 2, \ldots, N_A - 1\}$). This greedy policy will prioritize time constraints of high-priority applications.

Algorithm 1. Priority-Based Greedy Scheduling Algorithm

Input: Γ: An Application Set, N_A: The Number of Applications, $M^N_{WCET}[N_A][100]$: WCET Matrix of Applications with No-load, D_i Relative Deadline of App_i, P_i: Priority of App_i, j: Change Times of Workload, $\tau_i^{\delta_{i,j},L_{i,j}}$: Actual Execution Time of App_i with workload $L_{i,j}$ on $\delta_{i,j}$ vprocessors

Output: $\Delta[N_A]$: vprocessor Array for all applications, the element $\delta_{i,j+1} \in \Delta$ is the number of vprocessors allocated to App_i in the $(j+1)^{th}$ workload change

1: $P \leftarrow 0, C \leftarrow 0, \delta \leftarrow 0, R_\delta \leftarrow 0$;
2: **for** $App_i \in \Gamma$ **do**
3: **if** App_i is active **then**
4: $C_i^{\delta_{i,j},L_{i,j}} \leftarrow \textbf{SOFNN}(\delta_{i,j}, L_{i,j})$;
5: **if** $j == 0$ **then**
6: $r_i^{\delta_{i,j},L_{i,j}} \leftarrow C_i^{\delta_{i,j},L_{i,j}}$;
7: **else**
8: $r_i^{\delta_{i,j},L_{i,j}} \leftarrow C_i^{\delta_{i,j},L_{i,j}}\left(1 - \sum_{k=0}^{j-1} \frac{\tau_i^{\delta_{i,j},L_{i,k}}}{C_i^{\delta_{i,j},L_{i,k}}}\right)$;
9: **end if**
10: $\tau_i \leftarrow \sum_{k=0}^{j} \tau_i^{\delta_{i,k},L_{i,k}}$;
11: $C_i^{\delta_{i,j},N} \leftarrow M^N_{WCET}[i][\delta_{i,j}]$;
12: $C \leftarrow (D_i - \tau_i)\frac{C_i^{\delta_{i,j},N}}{r_i^{\delta_{i,j},L_{i,j}}}$;
13: $\delta_{i,j+1} \leftarrow \textbf{FINDMINDELTA}(C)$;
14: $\delta \leftarrow \delta + \delta_{i,j+1}$;
15: **if** $\delta \leq 100$ **then**
16: $\Delta[i] \leftarrow \delta_{i,j+1}$;
17: $P \leftarrow P + P_i$;
18: **else**
19: **for** $m = i$ to N_A **do**
20: $\Delta[m] \leftarrow 0$;
21: **end for**
22: Mark Γ is unschedulable;
23: **return** Δ;
24: **end if**
25: $\Delta[i] \leftarrow 0$;
26: **end if**
27: **end for**
28: **if** $\delta < 100$ **then**
29: $R_\delta \leftarrow 100 - \delta$;
30: **for** $App_i \in \Gamma$ **do**
31: **if** App_i is active **then**
32: $\Delta[i] \leftarrow \Delta[i] + \lfloor R_\delta \times \frac{P_i}{P} \rfloor$;
33: **end if**
34: **end for**
35: Mark Γ is schedulable;
36: **end if**
37: **return** Δ;

In Algorithm 1, we check all applications in Γ. If the App_i is inactive, the $\delta_{i,j+1}$ is set to zero which means no computing resource is allocated to App_i. Otherwise, it should adjust vprocessors for App_i when workload is changed.

In Step 4, the function *SOFNN* is to predict execution time of App_i with workload-related indicators $L_{i,j}$ and the number of vprocessors $\delta_{i,j}$ based on the proposed GPU performance model.

From Step 5 to Step 12, we obtain a value C that is the upper bound of $C_i^{\delta_{i,j+1},N}$ to satisfy the real-time requirement. Then, we find the minimum $\delta_{i,j+1}$ in M^N_{WCET} via the function *FINDMINDELTA* to ensure that the inequation $C_i^{\delta_{i,j+1},N} \leqslant C$ is satisfied.

From Step 14 to Step 24, it is necessary to compare the sum of allocated vprocessors with the entire computing resources. If the allocated vprocessors is less than or equal to 100, the $\Delta[i]$ is updated by $\delta_{i,j+1}$. Otherwise, the GPU scheduler sets 0 to all the rest applications and marks a scheduling failure. That means vprocessors will not be allocated for those applications until the higher priority application is finished.

From Step 28 to Step 36, when traversing all applications in Γ, we can get an array Δ with the minimum $\delta_{i,j+1}$ for active applications that meet the real-time requirement. If the sum of allocated vprocessors is less than 100, the GPU scheduler will reallocate the rest computing resources R_δ for all active applications proportionally. The proportion of each App_i is determined by its priority P_i and the sum of priorities of all active applications. This makes full use of GPU computing resources to improve system performance. In this case, Γ is marked as schedulable.

5.3 Scheduling Optimization

In the proposed real-time scheduling algorithm, each workload change leads to adjustments of computing resources for all active applications. This brings substantial scheduling overhead for the airborne embedded system. To make it work efficiently, we design two optimizations as follows.

1) Workload Degradation Checking (WDC). In the basic implementation, a workload change caused by both application starting and finishing would trigger a scheduling operation. This operation may reallocate vprocessors for all active applications. Obviously, if the workload change is due to application finishing, the rest applications are co-executing at a lower load level. This workload degradation will shorten the execution time for all rest applications. Therefore, if the application set Γ is schedulable at last period, it is also schedulable without any reallocation of vprocessors at current period. This optimization reduce the scheduling overhead of vprocessors reallocation when the workload is decreased.

2) Actual Execution Time Checking (AETC). This optimization is based on an observation from experiments. In the proposed scheduling algorithm, we first predict the execution time $C_i^{\delta_{i,j},L_{i,j}}$ of App_i when it runs concurrently with the stable workload $L_{i,j}$ on $\delta_{i,j}$ vprocessors via SOFNN model. Then, we calculate the remaining execution time $r_i^{\delta_{i,j},L_{i,j}}$ according to $C_i^{\delta_{i,j},L_{i,j}}$ and the actual execution time $\tau_i^{\delta_{i,j},L_{i,j}}$. The calculated remaining execution time $r_i^{\delta_{i,j},L_{i,j}}$ has a larger error when the App_i is just started since the predicted execution time $C_i^{\delta_{i,j},L_{i,j}}$ does not take workload change into account. However, the error continues to decrease over the actual execution time $\tau_i^{\delta_{i,j},L_{i,j}}$. We transform $\tau_i^{\delta_{i,j},L_{i,j}}$ to a standard execution time $\tau_i^{100,N}$, which represents the actual execution time that App_i runs individually without any workload. We define a normalized completion rate (denoted by θ), which is the ratio of $\tau_i^{100,N}$

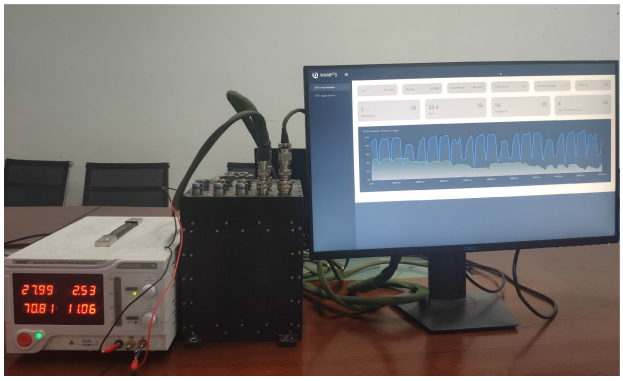


Fig. 8. The airborne embedded system.

to $C_i^{100,N}$. If θ is greater than a predefined threshold Θ_{th}, the error is small enough so that we can use $r_i^{\delta_{i,j-1},L_{i,j-1}}$ in place of $r_i^{\delta_{i,j},L_{i,j}}$. This optimization reduce the scheduling overhead of calculating the remaining execution time.

6 Evaluation

6.1 Experimental Setup

We implement a prototype of WAMP²S on a real airborne embedded system as shown in Fig. 8. The airborne embedded system is composed of a master control board with a 2.0GHz Intel i7-6822EQ CPU and 16GB main memory, a parallel processing board with a 3U VPX GPU card, a PCIe switch borad and an I/O control board. The detail hardware specifications of the GPU card is given in Table 2. The operating system is the Ubuntu18.04 with the Linux kernel 4.18.0.

We evaluate the proposed GPU performance model and scheduling algorithm on both simulated and realistic application sets. We select 10 applications from Rodinia benchmark [39] to form the simulated application set, which are listed in Table 3. These applications cover a wide range of execution patterns, such as computation intensive, global memory intensive, L2 cache intensive, shared memory intensive and texture memory intensive.

In Table 3, we give the measured WCET of each selected Rodinia application when it runs individually without any workload on the whole GPU. To analyze the schedulability of WAMP²S, we use *UUniSort* algorithm proposed in [40] to generate corresponding periods for applications with a desired overall utilization U

$$U = \sum_{i=0}^{N_A} \frac{C_i^{100,N}}{T_i}. \tag{14}$$

In the experiments, we adopt implicit deadline that the relative deadline D_i is equal to the period T_i.

TABLE 2
Specifications of the GPU Board

Parameter	Value	Parameter	Value
GPU	GTX1060	Core frequency	1506MHz
Architecture	Pascal	Memory frequency	8000MHz
CUDA Core	1280	Memory size	6GB DDR5
SM number	10	Memory bandwidth	192GB/s

TABLE 3
Simulated Application Set

App (Abbr.)	Domains	WCET (s)	Priority
Gaussian (GS)	Linear Algebra	0.261	10
Heart Wall (HW)	Medical Imaging	0.356	9
cfd (CFD)	Fluid Dynamics	2.081	8
kmeans (KM)	Data Mining	0.079	7
lavaMD (LM)	Molecular Dynamics	0.448	6
leukocyte (LC)	Medical Imaging	0.144	5
lud (LUD)	Linear Algebra	0.028	4
myocyte (MC)	Biological Simulation	0.531	3
srad (SRD)	Image Processing	0.301	2
streamcluster (ST)	Data Mining	1.444	1

The realistic application set of the airborne embedded system is shown in Table 4. It is composed of 5 applications: Behavior Decision (BD), Communication Management (CM), Mission Allocation (MA), Path Planning (PP) and Object Detection (OD). For sporadic application, the period is the inter-arrival time. All these applications need to use the GPU device to accelerate performance.

6.2 Experimental Results

6.2.1 Model Training and Comparison

The training phase of App_i in the simulated application set is conducted as described in Section 4.3.2. The workload is generated by different combinations of other applications, and runs continuously until App_i is finished. We collect over 4000 sample pairs $\langle S_m^k, S_e^k \rangle$ for one application. Each SOFNN model of an application is trained by 3000 sample pairs, and the rest samples are used to evaluate the accuracy of the workload-aware GPU performance model. Fig. 9 gives the training result of SOFNN for lavaMD application. There are a total of 55 fuzzy rules generated during the structure learning phase. Correspondingly, each workload-related indicator has 55 membership functions with parameters as shown in Figs. 9b and 9c.

We compare the proposed SOFNN model with other popular neural network model, including Back Propagation Neural Network (BPNN) and Deep Neural Network (DNN). The layers of the BPNN and DNN are 4 and 12 respectively. Fig. 10 shows the Root Mean Square Error (RMSE) and the computational overhead of BPNN, DNN and SOFNN model trained for the lavaMD application. We can observe that DNN has the highest predict accuracy, but the computation time is too long to be used in the GPU scheduler. The computational overhead of BPNN is almost as low as the SOFNN, however, the RMSE of BPNN is very high. By comprehensive comparison, the lightweight SOFNN is a proper model considering both accuracy and overhead.

6.2.2 Overall Accuracy

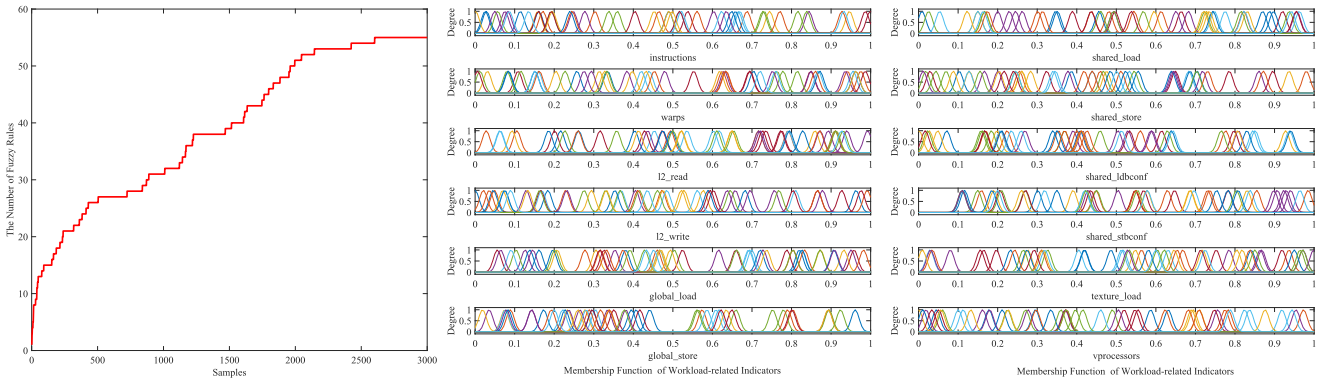
We choose the Mean Absolute Percentage Error (MAPE) as the accuracy metric which is defined as follows:

$$MAPE = \frac{1}{N_S} \sum_{k=1}^{N_S} \frac{|S_m^k - S_e^k|}{S_e^k} \times 100\%, \tag{15}$$

where S_m^k is the measured execution time via the monitor thread and S_e^k is the estimated execution time via the SOFNN model.

TABLE 4
Realistic Application Set

App (Abbr.)	Implementation	Type	WCET (s)	Period (s)	Priority
Behavior Decision (BD)	Hierarchical Reinforcement Learning	Periodic	0.405	1	10
Communication Management (CM)	Multi-Agent Reinforcement Learning	Periodic	1.712	5	8
Mission Allocation (MA)	Graph Neural Network	Sporadic	2.071	30	6
Path Planning (PP)	Parallel Genetic Algorithm	Sporadic	2.896	30	5
Object Detection (OD)	Deep Neural Network	Periodic	8.495	20	3



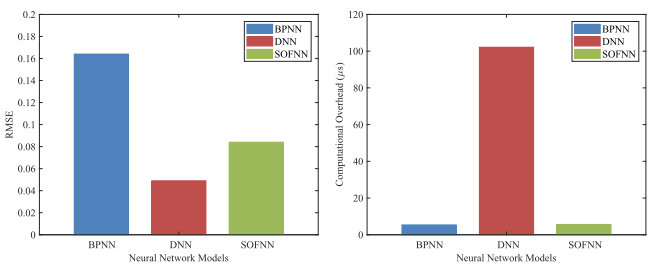
(a) Growth of number of fuzzy rules (b) Membership functions (indicators 1-6) (c) Membership functions (indicators 7-12)

Fig. 9. SOFNN training result.

In the evaluation, the workload still remains stable with the application being tested. From Fig. 11, the proposed GPU performance model can achieve a low average MAPE from 0.95% to 2.58%. Actually, the workload may change during the execution of applications. Therefore, we evaluate the model accuracy in a more practical way. For the application App_i, the monitor thread records the predict the execution time $C_i^{\delta_{i,j},L_{i,j}}$ by SOFNN and the actual execution time $\tau_i^{\delta_{i,j},L_{i,j}}$ when the j^{th} workload change occurs. Then, the estimated remaining execution time $r_i^{\delta_{i,j},L_{i,j}}$ can be calculated according to Eq. (11).

Fig. 12 gives the error rate of the estimated remaining execution time to the actual remaining execution time of application MC over the workload changes. In this case, application MC totally runs 685ms and experiences 4 workload changes. At the beginning of MC arrivals the GPU device, the GPU scheduler estimates the remaining execution time for it (first estimation). Then, the GPU scheduler will recalculate the remaining execution time when the workload change occurs (the rest 4 estimations). The first estimation has a large error rate because the GPU scheduler cannot foresee how many times does the workload change. But it is worth noting that the error rate decreases greatly with the actual execution time.

For consistency, we transform $C_i^{\delta_{i,j},L_{i,j}}$, $\tau_i^{\delta_{i,j},L_{i,j}}$ and $r_i^{\delta_{i,j},L_{i,j}}$ to $C_i^{100,N}$, $\tau_i^{100,N}$ and $r_i^{100,N}$ for App_i by linear mapping. Here, we define a normalized application completion rate (denoted by θ), which is the ratio of $\tau_i^{100,N}$ to $C_i^{100,N}$. To analyzing the relationship between the estimated error rate of remaining execution time and the actual execution time, we record over 10000 samples from the selected 10 applications with varied workload changes. Each sample is formed by the normalized application completion rate and the corresponding error rate. Fig. 13 illustrates the relationship. With this observation, we set a threshold Θ_{th} that ensure the estimated error rate is less than 1% when θ is greater than Θ_{th}. Thus, for an application App_i, the scheduling operation cannot be triggered if the workload change happens after the time $\Theta_{th} \times C_i^{100,N}$. For the simulated application set, the threshold Θ_{th} is 76.42%.



(a) Error Rate (b) Computational Overhead

Fig. 10. Comparison of different neural network models.

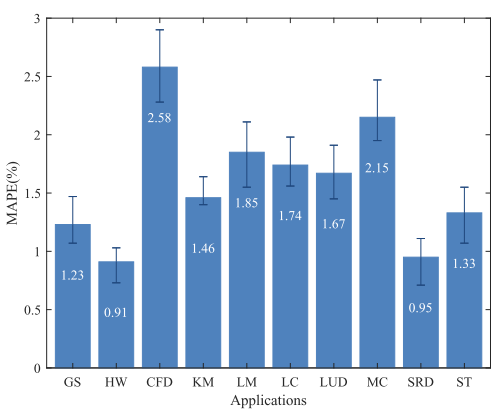


Fig. 11. MAPE of the simulated application set.

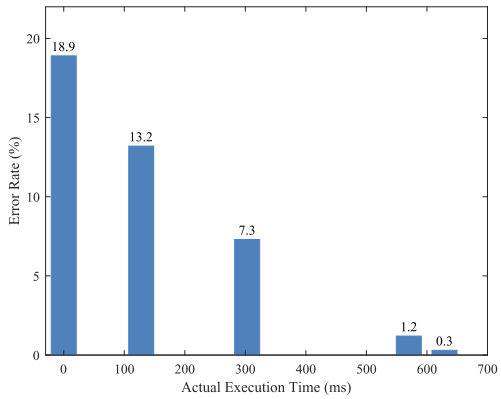


Fig. 12. Estimated error rate of MC over workload changes.

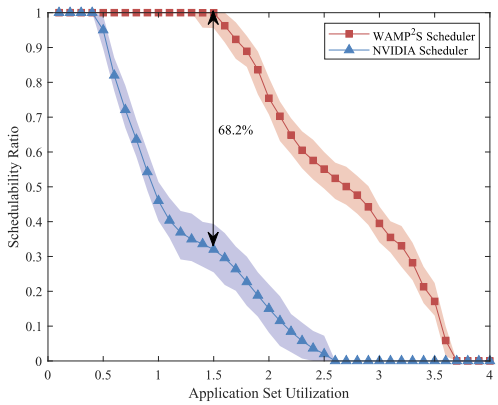


Fig. 14. Schedulability versus Utilization.

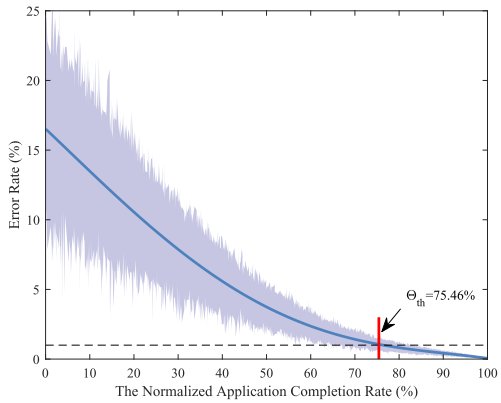


Fig. 13. Error rate over the application completion rate.

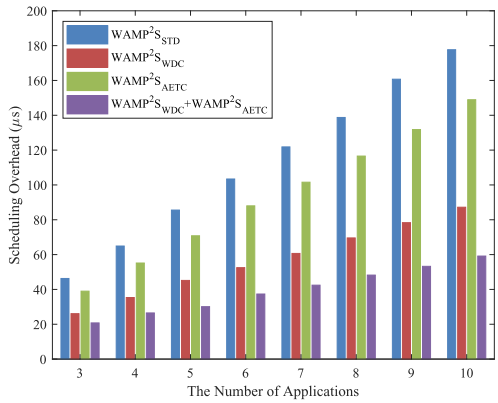


Fig. 15. Average scheduling overhead.

6.2.3 Schedulability

The schedulability experiments are conducted on different application sets generated by simulated Rodinia applications. As described in Section 6.1, periods of selected applications are generated via *UUniSort* algorithm according to a desired utilization for each experiment. Fig. 14 compares the schedulability ratio of application sets as a function of the overall utilization between the inherent NVIDIA scheduler and the proposed WAMP²S scheduler. Each point in Fig. 14 corresponds to 60 randomly generated application sets.

It is obviously that the schedulability ratio decreases with the increasing overall utilization of the whole application set, because the resource contention becomes more intense with co-executing applications. The schedulability ratio of the NVIDIA scheduler drops significantly when $U > 0.6$. The WAMP²S scheduler maintains 100% success rate until $U > 1.5$. At most 68.2% more applications are schedulable under the WAMP²S scheduler.

6.2.4 Scheduling Overhead

The scheduling overhead incurred by the WAMP²S scheduler running a scheduling operation. Such an operation is triggered when an existing application is finished or a new application is launched in the standard WAMP²S ($\text{WAMP}^2\text{S}_{\text{STD}}$) implementation. In Section 5.3, we design two optimized scheduling methods: WAMP²S with workload degradation checking ($\text{WAMP}^2\text{S}_{\text{WDC}}$) and WAMP²S with actual execution time checking ($\text{WAMP}^2\text{S}_{\text{AETC}}$).

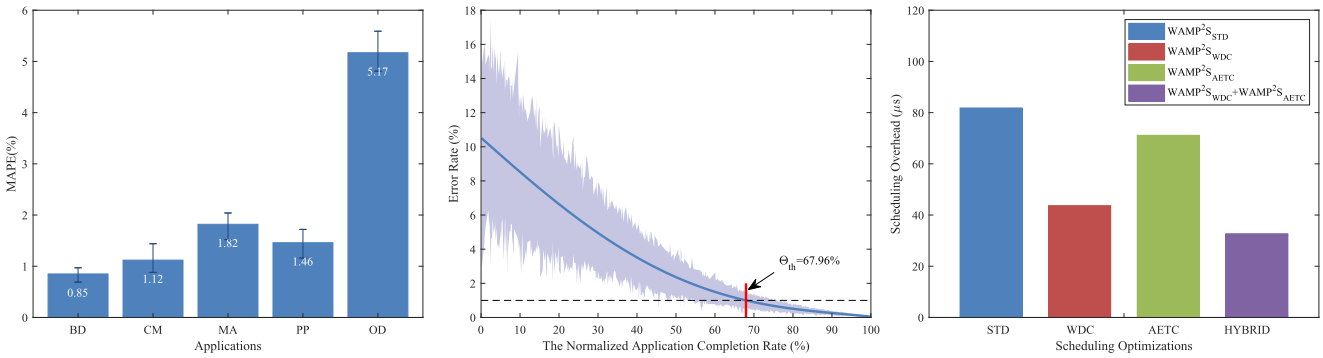
The number of applications N_A is set from 3 to 10. Then, the application set is formed randomly from Rodinia applications according to N_A. For each scheduling overhead evaluation, an application App_i releases a job $N_{J,i}$ times periodically. During one job execution, the scheduling operation is triggered $N_{S,j}$ times each of which has the scheduling overhead t_k. Thus, the average scheduling overhead of this experiment is calculated as follows:

$$T_o = \frac{1}{N_A}\sum_{i=1}^{N_A}\left(\frac{1}{N_{J,i}}\sum_{j=1}^{N_{J,i}}\sum_{k=1}^{N_{S,j}} t_k\right). \tag{16}$$

In the experiment, the cycles $N_{J,i}$ of App_i is set to be 200 times. We measured the average scheduling overhead for an application set based on 100 experiments. From Fig. 15, the scheduling overhead increases proportionally to the number of applications due to the growing probability of workload changes caused by more and more co-executing applications. Compared to $\text{WAMP}^2\text{S}_{\text{STD}}$, $\text{WAMP}^2\text{S}_{\text{WDC}}$ and $\text{WAMP}^2\text{S}_{\text{AETC}}$ reduce 48.5% and 16.2% average scheduling overhead, respectively. Moreover, the hybrid optimization reduces 63.3% average scheduling overhead. In a set with 10 applications, the average scheduling operation time for one job of an application is just 59.2μs.

6.3 Results of the Realistic Application Set

The results of the realistic application set is shown in Fig. 16. The MAPE of the realistic application set is ranging from 0.85% to 5.17%. The threshold Θ_{th} obtained from experiments



(a) MAPE of the realistic application set (b) Error rate over the application completion rate (c) Scheduling overhead

Fig. 16. Results of the realistic application set.

is 67.96%. The hybrid optimization reduces 60.1% average scheduling overhead and the average scheduling operation time is 32.7μs for one job of an application. To analyze the schedulability of WAMP^{2}S, we develop a monitoring software to record timing sequences of all applications. The overall utilization ($U = 1.34$) of the realistic application set can be derived from Table 4. Fig. 17 shows the basic information and timing diagram of one test runs. The scheduling success ratio has always been 100% in 8 long test runs (each test lasts 30 mins).

6.4 Discussion

WAMP^{2}S considers the impact of co-execution workload on the execution time estimation and provides a software-only approach for preemption support. It does not need any source code analysis or transformation. Unlike previous work on preemptive GPU scheduling that save and resume the context at scheduling point, the pseudo-preemptive scheduling can provide the approximate preemption by dynamically adjusting GPU computing resources for active applications. Compared with the state-of-the-art preemptive GPU scheduling algorithm for embedded system [32], the average scheduling overhead of WAMP^{2}S in a set of 10 applications is just 59.2μs without any memory copy operation. The pre-preemption overhead of scheduling algorithm proposed in [32] is less than 10μs in a set of 11 applications, but the overhead of saving and resuming kernel context (snapshotting operation) is more than 300μs per MB.

But there are still some limitations in the prototype WAMP^{2}S scheduler. Since it is specifically designed for the airborne embedded system where the application set is usually small and fixed, WAMP^{2}S can constantly improve the prediction accuracy of GPU performance model by training with sufficient data. However, WAMP^{2}S is not applicable for the general-purpose system such as data center or cloud platform, which has a large and changing application set. Another limitation is that the proposed SOFNN model does not consider the performance difference among various GPU devices. We will add some GPU performance-related parameters as the input variables of SOFNN model to deal with this problem in our future work.

Fig. 17. The monitoring software of WAMP^{2}S.

7 Conclusion

In this paper, we implement a real-time GPU scheduler, WAMP^{2}S, for the airborne embedded system. First, a workload-aware GPU performance model is proposed to predict the execution time of applications under co-execution circumstance. The model collects workload-related indicators at runtime and estimates the execution time of the application based on SOFNN. Then, a pseudo-preemptive GPU scheduler is designed to deal with the real-time scheduling in the embedded system. The pseudo-preemptive scheduler dynamically adjusts computing resources for active applications according to the priority and deadline. Finally, we implement a prototype of WAMP^{2}S on a real airborne embedded platform and evaluate the proposed model and algorithm on both simulated and realistic application sets. Experimental results illustrate that the proposed WAMP^{2}S scheduler can achieve an average predicted error of 0.85%∼5.12% and at most 68.2% more scheduling success ratio than the NVIDIA scheduler.

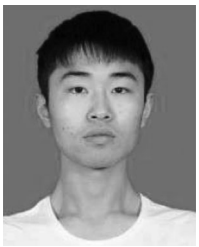
References

[1] R. L. Davidson and C. P. Bridges, "Error resilient GPU accelerated image processing for space applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 1990–2003, Sep. 2018.

[2] S. Liu, J. Chen, Y. Ai and S. Rahardja, "An optimized quantization constraints set for image restoration and its GPU implementation," *IEEE Trans. Image Process.*, vol. 29, pp. 6043–6053, 2020.
[3] A. M. Sainju, D. Aghajarian, Z. Jiang, and S. Prasad, "Parallel grid-based colocation mining algorithms on GPUs for big spatial event data," *IEEE Trans. Big Data*, vol. 6, no. 1, pp. 107–118, Mar. 2020.
[4] B. Zhu, Y. Jiang, M. Gu, and Y. Deng, "A GPU acceleration framework for motif and discord based pattern mining," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 1987–2004, Aug. 2021.
[5] M. Martínez-Zarzuela, C. Gómez, F. J. Díaz-Pernas, A. Fernández, and R. Hornero, "Cross-approximate entropy parallel computation on GPUs for biomedical signal analysis. Application to MEG recordings," *Comput. Methods Programs Biomed.*, vol. 112, no. 1, pp. 189–199, 2013.
[6] C.-L. Hung and C.-Y. Lin, "GPU-based texture analysis approach for mammograms institute of biomedical informatics," in *Proc. IEEE Int. Conf. Bioinformat. Biomed.*, 2020, pp. 2183–2186.
[7] João Báuto, A Canelas, R. Neves, and N. Horta, "Parallel SAX/GA for financial pattern matching using NVIDIAs GPU," *Expert Syst. Appl.*, vol. 108, no. 1, pp. 77–88, 2018.
[8] S. Scheidegger, D. Mikushin, F. Kubler and O. Schenk, "Rethinking large-scale economic modeling for efficiency: Optimizations for GPU and Xeon Phi Clusters," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 610–619.
[9] R. Nath and D. Tullsen, "The CRISP performance model for dynamic voltage and frequency scaling in a GPGPU," in *Proc. 48th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2015, pp. 281–293.
[10] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proc. 21st IEEE Int. Symp. High Perform. Comput. Archit.*, 2015, pp. 564–576.
[11] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, "A performance model for GPUs with caches," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1800–1813, Jul. 2015.
[12] X. Wang, K. Huang, A. Knoll, and X. Qian, "A hybrid framework for fast and accurate GPU performance estimation through source-level analysis and trace-based simulation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 506–518.
[13] J. Lew *et al.*, "Analyzing machine learning workloads using a detailed GPU simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 151–152.
[14] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2019, pp. 293–303.
[15] J. Wu, X. Yang, Z. Zhang, G. Chen, and R. Mao, "A performance model for GPU architectures that considers on-chip resources: Application to medical image registration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 1947–1961, Sep. 2019.
[16] Q. Wang and X. Chu, "GPGPU performance estimation with core and memory frequency scaling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 12, pp. 2865–2881, Dec. 2020.
[17] X. Wang, X. Qian, A. Knoll, and K. Huang, "Efficient performance estimation and work-group size pruning for OpenCl kernels on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 5, pp. 1089–1106, May 2020.
[18] NVIDIA KEPLER GK110 Next-Generation CUDA Compute Architecture. [Online]. Available: https://www.nvidia.cn/content/dam/en-zz/Solutions/Data-Center/documents/NV-DS-Tesla-KCompute-Arch-May-2012-LR.pdf
[19] G. A. Elliott and J. H. Anderson, "Real-world constraints of GPUs in real-time systems," in *Proc. 17th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2011, pp. 48–54.
[20] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Annu. Tech. Conf.*, 2011, Art. no. 2.
[21] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Proc. 32nd IEEE Real-Time Syst. Symp.*, 2011, pp. 57–66.
[22] S. Kato, M. McThrow, C. Maltzahn, and S. A. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 401–412.
[23] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *Proc. 34th IEEE Real-Time Syst. Symp.*, 2013, pp. 33–44.
[24] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. 41st ACM/IEEE Int. Symp. Comput. Archit.*, 2014, pp. 193–204.
[25] J.J.K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. 20th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2015, pp. 593–606.
[26] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous multikernel GPU: Multi-tasking throughput processors via fine-grained sharing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 358–369.
[27] J. Zhong and B. He, "Kernelet: High-throughput GPU kernel executions with dynamic slicing and scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1522–1532, Jun. 2014.
[28] H. Zhou, G. Tong, and C. Liu, "GPES: A preemptive execution system for GPGPU computing," in *Proc. 21st IEEE Real-Time Embedded Technol. Appl. Symp.*, 2015, pp. 87–97.
[29] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 3–16.
[30] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling flexible and efficient preemption on GPUs," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2017, pp. 483–496.
[31] C. Hartmann, U. Margull, "GPUart-An application-based limited preemptive GPU real-time scheduler for embedded systems," *J. Syst. Archit.*, vol. 97, pp. 304–319, 2019.
[32] H. Lee, H. Kim, C. Kim, H. Han, and E. Seo, "Idempotence-based preemptive GPU Kernel scheduling for embedded systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 70, no. 3, pp. 332–346, Mar. 2021.
[33] L. Shi, H. Chen, J. Sun and K. Li, "vCUDA: GPU-accelerated high-performance computing in virtual machines," *IEEE Trans. Comput.*, vol. 61, no. 6, pp. 804–816, Jun. 2012.
[34] J. Gu, S. Song, Y. Li and H. Luo, "GaiaGPU: Sharing GPUs in container clouds," in *Proc. IEEE Int. Conf. Parallel Distrib. Process. Appl., Ubiquitous Comput. Commun., Big Data Cloud Comput., Social Comput. Netw., Sustainable Comput. Commun.*, 2018, pp. 469–476.
[35] Kubelet. Accessed: Aug. 23, 2018. [Online]. Available: https://kubernetes.io/docs/concepts/overview/
[36] CUDA Profiling Tool Interface, "CUPTI User's Guide," 2020. [Online]. Available: https://docs.nvidia.com/cupti/pdf/Cupti.pdf
[37] Y. Yao *et al.*, "Brief industry paper: Workload-aware GPU performance estimation in the airborne embedded system," in *Proc. 27th IEEE Real-Time Embedded Technol. Appl. Symp.*, 2021, pp. 417–420.
[38] Y. Yao, K. Zhang, and X. Zhou, "Implement TSK model using a self-constructing fuzzy neural network," in *Proc. WRI Global Cong. Intell. Syst.*, 2009, pp. 479–483.
[39] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.
[40] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Syst.*, vol. 30, no. 1, pp. 129–154, 2005.

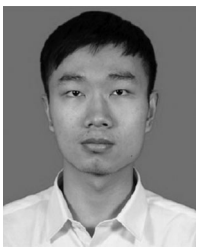
Yuan Yao (Member, IEEE) received the BS, MS, and PhD degrees in computer science from Northwestern Polytechnical University, Xi'an, China, in 2007, 2009 and 2015, respectively. He is currently an associate professor with the School of Computer Science, Northwestern Polytechnical University. His research interests include real-time embedded system, distributed and parallel computing, and system software.

Shuangyang Liu received the BEs degree from the Hubei University of Arts and Science in 2019. She is currently working toward the master's degree with the Department of Computer Science, Northwestern Polytechnical University. Her research interests include parallel computing and GPU real-time scheduling.



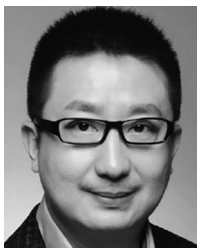
Sikai Wu received the BEs degree from the Huaiyin Institute of Technology in 2019. He is currently working toward the master's degree with the Department of Computer Science, Northwestern Polytechnical University. His research interests include GPU virtualization and GPU resource management.

Jinyu Wang received the BS degree from the Taiyuan University of Science and Technology, Taiyuan, China, in 2020. She is currently working toward the master's degree with the School of Computer Science, Northwestern Polytechnical University. Her research interests include on real-time embedded system and GPU power-efficient computing.



Jinting Ni received the BEs degree from the Huaiyin Institute of Techology in 2020. He is currently working toward the master's degree with the Department of Computer Science, Northwestern Polytechnical University. His research interests include real-time embedded system and resource management.

Gang Yang (Member, IEEE) received the BE degree from the School of Automation from Second Artillery Engineering College of PLA, Xi'an, China, in 1998 and the MS and PhD degrees in computer science from Northwestern Polytechnical University, Xi'an, China, in 2002 and 2006, respecively. He is currently a professor with the School of Computer Science, Northwestern Polytechnical University, Xi'an, China. His research interests include distributed computing systems, intelligent swarm systems, and cyber-physical systems.



Yu Zhang (Member, IEEE) received the PhD degree in computer science from Northwestern Polytechnical University, Xi'an, China, and RMIT University, Melbourne, VIC, Australia. He was a scholarship researcher with The University of Melbourne. He is currently an associate professor with the School of Computer Science, Northwestern Polytechnical University. His research interests include protocol design, wireless sensor networks, mobile computing, Internet of Things, and human-CPS resource management. He is a member of ACM.