SHAPE: Scheduling of Fixed-Priority Tasks on Heterogeneous Architectures with Multiple CPUs and Many PEs

Yuankai Xu*[1], Tiancheng He*[1], Ruiqi Sun[1], Yehan Ma[1], Yier Jin[2], An Zou[1]
[1]Shanghai Jiao Tong University, [2]University of Science and Technology of China

ABSTRACT

Despite being employed in burgeoning efforts to accelerate artificial intelligence, heterogeneous architectures have yet to be well managed with strict timing constraints. As a classic task model, multi-segment self-suspension (MSSS) has been proposed for general I/O-intensive systems and computation offloading. However, directly applying this model to heterogeneous architectures with multiple CPUs and many processing units (PEs) suffers tremendous pessimism. In this paper, we present a real-time scheduling approach, SHAPE, for general heterogeneous architectures with significant schedulability and improved utilization rate. We start with building the general task execution pattern on a heterogeneous architecture integrating multiple CPU cores and many PEs such as GPU streaming multiprocessors and FPGA IP cores. A real-time scheduling strategy and corresponding schedulability analysis are presented following the task execution pattern. Compared with state-of-the-art scheduling algorithms through comprehensive experiments on unified and versatile tasks, SHAPE improves the schedulability by 11.1% - 100%. Moreover, experiments performed on the NVIDIA GPU systems further indicate up to 70.9% of pessimism reduction can be achieved by the proposed scheduling. Since we target general heterogeneous architectures, SHAPE can be directly applied to off-the-shelf heterogeneous computing systems with guaranteed deadlines and improved schedulability.

CCS CONCEPTS

• **Computer systems organization → Real-time systems**.

KEYWORDS

Real-time Scheduling, Heterogeneous Computing

1 INTRODUCTION

Computing systems, both for embedded and cloud applications, are increasingly adopting heterogeneous architectures to handle the growing demand for high-performance and energy-efficient computing in emerging artificial intelligence (AI) workloads [1]. In many real-world applications, such as autonomous driving [2] and robotics [3], these AI tasks require real-time execution, which demands efficient task scheduling and allocation to meet strict timing constraints. To address this, heterogeneous computing platforms, such as GPU servers [4], Xilinx UltraScale [5], and TI Keystone II [6], integrate CPU cores and parallel processing elements (PEs), such as GPU Streaming Multiprocessors or FPGA IP cores, to leverage the strengths of each type of processor.

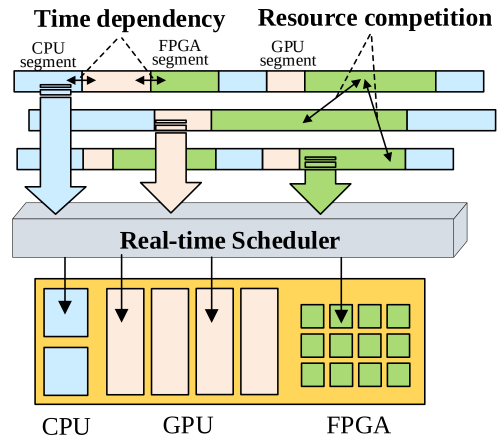


Figure 1: Real-time scheduling of parallel tasks on the heterogeneous architecture.

Tasks running on heterogeneous computing platforms typically have a segmented structure, as illustrated in Fig. 1. To optimize performance and energy efficiency, serial computation segments are usually allocated to CPU cores, while data-parallel segments are offloaded to PEs, which are known as GPU or FPGA segments. However, this interleaved execution pattern can cause dependencies and competition between parallel tasks, leading to complex scheduling challenges [7]. The resulting task execution pattern makes it difficult to meet both timing constraints and high resource utilization rates simultaneously, requiring innovative scheduling techniques to address the scheduling problem [8, 9]. Additionally, as the number of CPU and PE cores, and corresponding computation segments, increases, significant reductions in schedulability are commonly observed [10] [11]. Therefore, effective scheduling algorithms and resource management techniques are critical to enable efficient and reliable execution of AI tasks on heterogeneous computing platforms.

Targeting general heterogeneous architectures with multiple CPU and many PEs, this paper presents SHAPE, a real-time scheduling strategy, and corresponding response time analysis with improved schedulability. Through the extensive experiments by numerical simulation and real GPU systems, the SHAPE significantly

*Authors contributed equally to this research. An Zou is the corresponding author. This research project is supported by Huawei Technologies, NSFC 62103268, and Shanghai Chenguang Program 21CGA11.

Submission to ICCAD, Oct 30–Nov 3, 2022, San Diego, USA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

improves the schedulability by 11.1%-100% compared with previous works on the heterogeneous computing platforms. Overall, the main contributions of this paper are three-fold.

- Starting with modeling general heterogeneous computing architectures which have multi-core CPU and many-core PEs, a scheduling strategy with federated and fixed-priority scheduling is presented.
- An end-to-end response time analysis is performed, leveraging the workload function in self-suspension models. The essential properties in the analysis enables the scheduling to be compatible with optimal priority assignment.
- Extensive numerical and real CPU-GPU experiments are conducted to demonstrate that the proposed approach can improve the schedulability on unified and versatile machine learning tasks, and effectively reduce the pessimism.

2 BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Heterogeneous Architectures. Heterogeneous systems, consisting of CPU and parallel PEs, are becoming more prevalent in embedded computing areas (e.g., NVIDIA Jetson Series) and high-performance computing communities (e.g., Oak Ridge's Titan supercomputer). Such systems can offer higher performance at lower energy costs than homogeneous systems. CPU cores are the central controller of heterogeneous systems, and the PEs are regarded as the auxiliary devices to accelerate parallel arithmetic operations. There are three mainstream parallel processing elements in modern computing systems: GPU, FPGA, and Accelerators. In general, for an application running on a heterogeneous system, the CPU takes charge of the I/O, and serial computation, while the parallel executions are offloaded to the parallel PEs. Many scheduling strategies for heterogeneous architectures mainly treat the PEs as an inseparable component. In recent years, researchers and processor vendors have gradually supported the spatial partitioning of the PEs for concurrent applications. For example, Multi-Process Service (MPS) [12] and Multi-Instance GPU (MIG) [4] have been introduced by NVIDIA to support multiple task concurrency with assigned numbers of PEs to each task. AMD released open-source software support for hardware partitioning, which has the potential to accelerate and aid the long-term viability of real-time GPU research [13, 14]. To reap the benefits of this fine-grained partitioning on PEs, this paper proposes SHAPE on heterogeneous architectures with multiple CPU cores and many PEs which can be quantitatively assigned to different tasks.

2.1.2 Workload Function of Multi-segment Self-Suspension Models. One of the classic task models on heterogeneous architecture is the multi-segment self-suspension (MSSS) model. In this model, a task τ_i has m_i execution segments and $m_i - 1$ suspension segments between the execution segments. So task τ_i with deadline D_i and period T_i is expressed as a 3-tuple

$$\tau_i = \left((L_i^0, S_i^0, L_i^1, ..., S_i^{m_i-2}, L_i^{m_i-1}), D_i, T_i\right), \tag{1}$$

where L_i^j and S_i^j are the lengths of the j-th execution and suspension segments, respectively. $[\check{S}_i^j, \hat{S}_i^j]$ represents the upper and lower bounds of the suspension length S_i^j. $\hat{L}_i^j$ is the upper bound on the length of the execution segment L_i^j. From the CPU perspective, the execution segments L_i^j are the CPU segments. While the suspension segments S_i^j are the workload offloaded to the PEs, which behave like suspension.

The workload function $W_i(t)$ is widely used in the self-suspension model, which models the workload of task i given a time length of t. Bletsas et al. [15] summarize the workload functions used in self-suspension model. One [16] of these workload functions is summarized below and utilized in this work.

Lemma 2.1. *The following workload function $W_i^h(t)$ bounds on the maximum amount of execution that task τ_i can perform during an interval with a duration t and a starting segment L_i^h,*

$$\begin{aligned} W_i^h(t) = & \sum_{j=h}^{l} \hat{L}_i^{j \bmod m_i} + \\ & \min\left(\hat{L}_i^{(l+1) \bmod m_i}, t - \sum_{j=h}^{l}\left(\hat{L}_i^{j \bmod m_i} + S_i(j)\right)\right), \end{aligned} \tag{2}$$

where l is the maximum integer satisfying the following condition

$$\sum_{j=h}^{l}\left(\hat{L}_i^{j \bmod m_i} + S_i(j)\right) \leqslant t,$$

and $S_i(j)$ is the minimum inter-arrival time between execution segments L_i^j and L_i^{j+1}, which is defined by

$$S_i(j) = \begin{cases} \check{S}_i^{j \bmod m_i} & \text{if } j \bmod m_i \neq (m_i - 1) \\ T_i - D_i & \text{else if } j = m_i - 1 \\ T_i - \sum_{j=0}^{m_i-1} \hat{L}_i^j - \sum_{j=0}^{m_i-2} \check{S}_i^j & \text{otherwise.} \end{cases}$$

2.2 Related Work

Based on the utilization of processing elements (PEs), the architecture for real-time scheduling on heterogeneous computing falls into three categories: treating the heterogeneous processing elements as a non-preemptive entirety, using a software approach to enable preemption of the heterogeneous processing elements, and spatially partitioning the heterogeneous processing elements to many individual hardware resources.

The original real-time scheduling on heterogeneous architectures mainly treats the heterogeneous PEs as a non-preemptive entirety. For example, in CPU-GPU scheduling, Kato et al. [17] introduced a priority-based scheduler. Elliott proposed shared resources and containers for integrating GPU and CPU scheduling [18] and GPUSync [19] for managing multi-GPU soft real-time systems with flexibility, predictability, and parallelism. Golyanik et al. [20] described a scheduling approach based on time-division multiplexing. S^3DNN [21] optimized the execution of DNN GPU workloads in a real-time multi-tasking environment through scheduling the GPU kernels. Common and significant advantages of these approaches are their generality and ease of use. Since they do not require any hardware modifications, they can be directly applied to the off-the-shelf heterogeneous computing platforms. However, these methods based on the non-preemptive entirety suffer a low schedulability because a higher priority task may be blocked by the bulky segments from lower priority tasks.

To overcome the limitation of blocking, many works extend the PEs with the preemption function [22, 23]. For example, Park et al. [24], Basaran et al. [25], Tanasic et al. [26], and Zhou et al. [27] proposed architecture extensions with hardware and software codesigns to improve the preemption and tested on the GPU simulators. The Effisha framework in [28] introduced software techniques without any hardware modification to support kernel preemption at the end of any arbitrary thread block. By mapping the schedulability problem to the reachability problem in timed automata, Yalcinkaya et al. [29] proposes an exact schedulability test for self-suspension tasks with fixed preemption points. However, the software and hardware design overhead for preemption prevents its wide adoption in many PEs, especially for the consideration of low cost and high performance.

Partitioning is another direction to support a flexible task execution on the PEs with low design costs [30]. In the aspect of hardware partitioning, real-time scheduling algorithms are presented by researchers worldwide. With the MSSS model and the workload functions, Huang et al. [9] presented a scheduling algorithm and response time analysis for uni-core CPU based heterogeneous architectures and achieved the tightest response time analysis; Saha et al. [11] introduced a software-hardware solution for efficient spatial-temporal scheduling for GPU; and Zou et al. [31] developed a scheduling mechanism for the heterogeneous systems with one CPU, one memory engine, and many GPU cores. Alongside the workload function, Patel et al. [10] extended the existing Multiprocessor Priority Ceiling Protocol (MPCP) schedulability analysis for the tasks with the MSSS model. While these techniques guarantee task deadlines, their pessimism significantly limits the hardware resource utilization rate. In this paper, we present a real-time scheduling strategy and response time analysis with superiorly improved schedulability and reduced pessimism, validated by numerical simulation and real CPU-GPU systems.

3 SYSTEM MODEL AND SCHEDULE STRATEGY

3.1 System Model and Notations

In this paper, we consider a general heterogeneous architecture with N_{CPU} CPU cores and N_{PE} processing elements (PEs). The heterogeneous architecture executes a set of n independent parallel real-time tasks $\tau = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$. The ith task τ_i is composed of M_i CPU segments separated by $M_i - 1$ PE segments. The task τ_i has its deadline D_i and release period T_i. A CPU segment is eligible to execute only after the completion of its previous PE segment and vice versa. Therefore, task τ_i can be characterized by 3 tuples,

$$\tau_i = \big((CL_i^0, PL_i^0, CL_i^1, PL_i^1, ..., PL_i^{M_i-2}, CL_i^{M_i-1}), T_i, D_i\big), \quad (3)$$

where CL_i^j denotes the length of the $j+1$th CPU segment and PL_i^j denotes the length of the $j+1$th PE segment in task τ_i. Each task has a priority p_i and the CPU and PE segments in the task inherit the task's priority.

For the off-the-shelf heterogeneous computing system, the CPU cores are mostly with either x86 or ARM architecture, where a preemptive execution manner is generally supported in the CPU cores. Since CPU segments take charge of the I/O and control functions with rare parallel executions, every serial CPU segment only takes one CPU core to run, even more CPU cores available.

The PEs, such as GPUs [32] and other machine learning accelerators [33] naturally have parallel architectures. Since the PE segments are parallel operations, they are naturally evenly distributed on the PEs. As the zero copy [34] and unified memory [35] are widely deployed in heterogeneous architectures, the time for copying data from CPU cores to PEs is included in the PE execution time. Given N_{PE} PEs, the execution time PT of a PE segment PL follows the Amdahl and Gustafson's law [36],

$$PT = \frac{PL}{1 - P + N_{PE}P}, \quad (4)$$

where P is the proportion of the PE segment that can be executed in parallel, and $1 - P$ is the proportion that remains serial, such as copying data from CPU cores to PEs. Although preemption are gradually supported in advanced PEs such as NVIDIA GPU streaming multiprocessors, it is rarely available in most PEs such as FPGA IP cores or digital signal processor (DSP). Therefore, we assume the segments run on the PEs in a non-preemptive manner.

The task parameters, such as CL, PL, and P for every segment, can be profiled ahead of scheduling with the task worst-case execution time (WCET) in the above models. The actual execution time on hardware is equal to or smaller than the WCET. A shorter actual execution time in a conventional homogeneous computing system will not invalidate the model and analysis derived with WCET. However, in the heterogeneous computing system, a faster execution time will aggravate the workload function (described by Eq. (2)) as the following segment from the same task will be ready and begin to execute earlier [16]. Therefore, in this work, we make the actual execution time consistent with the WCET, similar to [8]. This can be implemented in the tasks by adding an elastic spinning waiting until the WCET is reached. Note that there still exists pessimism with the WCET-based modeling. A tighter WCET modeling is the eternal goal of researchers [37, 38].

3.2 Spatial and Temporal Scheduling

The key challenge of deriving the end-to-end scheduling algorithm for heterogeneous tasks is to simultaneously deal with *the dependence between segments in one task* and *the competition on the limited hardware resource from different tasks*. This section introduces the scheduling algorithm for parallel tasks on heterogeneous architectures, targeting the natural properties of serial execution on CPU cores and parallel execution on PEs.

We propose a scheduling strategy integrating the temporal access to the CPU cores and spatial partitioning for the PEs. For spatial partitioning, the N_{PE} PEs are partitioned to n groups, group i has N_{PE_i} PEs dedicated to the ith task. The partitioning and response time analysis will follow the federated scheduling. For temporal access, the access to N_{CPU} CPU cores from the CPU segments will follow a preemptive fixed-priority manner. Therefore, the end-to-end real-time schedule strategy coordinates a grid search on PEs spatial partitioning and following CPU core temporal access. The schedulability test will pass when a schedulable case is found following schedulability analysis in Section 4. In this paper, we restrict our attention to constrained-deadline tasks, where $D_i \leqslant T_i$, and tasks with fixed task-level priorities, where each task is associated with an optimal assigned priority detailed in Section

4.2. More precisely, when making scheduling decisions on CPU segments, the system always selects the segment with the highest priority among all available (ready) segments for that resource to execute. Of course, a task segment only becomes available if all the previous segments of that task have been completed.

4 SCHEDULABILITY ANALYSIS

4.1 End-to-end Response Time

Following the scheduling strategy in Section 3, N_{PE_i} of PEs are allocated to the task τ_i via the grid search on PE spatial partition.

Lemma 4.1. *Given N_{PE_i} of PEs allocated for task τ_i, the response time PT_i^j for the $j+1$th PE segment in task τ_i is calculated by*

$$PT_i^j = \frac{PL_i^j}{1 - P_i^j + N_{PE_i} P_i^j}, \tag{5}$$

where P_i^j is the proportion of the PE segment that can be executed in parallel, and $1 - P_i^j$ is the proportion that remains serial.

Proof. In each grid searched partitioning, N_{PE_i} numbers of PEs are allocate delicately to each task τ_i, such that the PE segments in task τ_i can start executing immediately after the completion of the ahead CPU segment CL_i^j. Therefore, each the response time PT_i^j of the PE segment with a length of PL_i^j follows the the Amdahl and Gustafson's law in Section 3. □

In this way, the mapping and execution of PE segments to PE hardware are explicitly controlled. Furthermore, tasks do not need to compete for PEs, so there is no blocking time on the non-preemptive PEs. Therefore, the interference between different PE segments is minimized, and the response times of PE segments are more predictable. After knowing the response time of PE segments, the task model in Eq. (3) will be updated with

$$\tau_i = \big((CL_i^0, PT_i^0, CL_i^1, PT_i^1, ..., PT_i^{M_i-2}, CL_i^{M_i-1}), T_i, D_i\big). \tag{6}$$

In this updated model, the adjacent CPU segments CL_i^j and CL_i^{j+1} in task τ_i are separated by a pre-determined time interval PT_i^j, which is the response time of PE segments. These CPU segments from parallel tasks inherit the task priority and execute on N_{CPU} CPU cores. The optimal priority assignment for each task is further discussed in Section 4.2. For a convenient analysis, we rearrange the task order according to the task priorities. The task with highest priority is numbered task τ_0 and the task with the ith highest priority is numbered task τ_{i-1}.

Definition 4.1 (Workload Function)**.** *The workload function W_i^j for the task τ_i starting from the CPU segment CL_i^j is defined as follows:*

$$W_i^j(t) = \sum_{j=h}^{l} CL_i^{j \bmod m_i} + \min\Big(CL_i^{(l+1) \bmod m_i}, t - \sum_{j=h}^{l} \big(CL_i^{j \bmod m_i} + PT_i(j)\big)\Big), \tag{7}$$

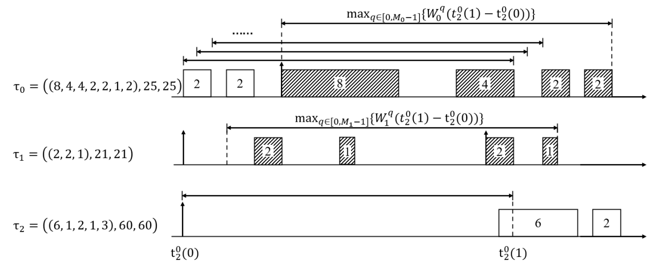


Figure 2: Example of the task τ_i that has been executed for 1 unit after it is released.

where l is the maximum integer satisfying the following condition:

$$\sum_{j=h}^{l} \big(CL_i^{j \bmod m_i} + PT_i(j)\big) \leqslant t,$$

and $PT_i(j)$ is the interval-arrival time between execution segments CL_i^j and CL_i^{j+1}, which is defined by

$$PT_i(j) = \begin{cases} PT_i^{j \bmod m_i} & \text{if } j \bmod m_i \neq (m_i - 1) \\ T_i - D_i & \text{else if } j = m_i - 1 \\ T_i - \sum_{j=0}^{m_i-1} CL_i^j - \sum_{j=0}^{m_i-2} PT_i^j & \text{otherwise.} \end{cases} \tag{8}$$

Following the Eq. (2) and Lemma 2.1 in the background section, the workload function W_i^j is an upper bound on the amount of CPU workload that task τ_i can generate during any time interval of t starting from the CPU segment CL_i^j.

Lemma 4.2. *Given the task τ_i released at time $t_i^0(0)$ (i.e., the first CPU segment CL_i^0 in task τ_i released at time $t_i^0(0)$), the CPU segment CL_i^0 has been executed for at least 1 unit at time $t_i^0(1)$, where $t_i^0(1)$ is the minimal integer satisfying the following condition*

$$\sum_{h \in hp(i)} max_{q\in[0\ M_h-1]}\{W_h^q(t_i^0(1) - t_i^0(0))\} < N_{CPU} * \big(t_i^0(1) - t_i^0(0)\big), \tag{9}$$

where $hp(i)$ is the group of tasks that have a higher priority than task τ_i and M_h is the total number of CPU segments in task τ_h.

Proof. In the duration of time interval $t_i^0(1) - t_i^0(0)$, N_{CPU} cores in the heterogeneous computing platform can process $N_{CPU} * (t_i^0(1) - t_i^0(0))$ units of workload, which is the expression on the right of the inequality. The CPU segments access the CPU cores in a fixed-priority and preemptive manner. To calculate the time upper-bound when the task τ_i has finished 1 unit of workload after release, we only need to account for the workload from the tasks with higher priorities than i, which is noted by $hp(i)$. By the definition 4.1, given the time interval of $t_i^0(1) - t_i^0(0)$ which starts from CPU segment CL_h^j, the CPU workload from the task τ_h is upper-bounded by the workload function W_h^j. In the run-time, the time interval of $t_i^0(1) - t_i^0(0)$ may starts from any CPU segment in task τ_h. Therefore, the worst-case (maximum) workload from task h can only be quantified by picking the maximum workload from the time interval of $t_i^0(1) - t_i^0(0)$ starting from any CPU segment CL_h^j in task τ_h i.e., $max_{q\in[0\ M_h-1]}\{W_h^q(t_i^0(1) - t_i^0(0))\}$. Therefore, the workload from higher-priority suspending tasks can be thereafter bounded by the summation of $max_{q\in[0\ M_h-1]}\{W_h^q(t_i^0(1)-t_i^0(0))\}$ for the tasks τ_h where $h \in hp(i)$. When the total workload from higher-priority suspending tasks is less than the workload CPU can process from time $t_i^0(0))$ to time $t_i^0(1)$, the task τ_i can be executed for at least 1 unit. The above proof is built on the fact that the CPU segment is serial computation, i.e., the CPU segment from one task can only generate 1 unit of workload at 1 unit of time. ◻

Case Study I A case study to calculate the time when the task τ_i has been executed for 1 unit after it is released is presented in Fig. 2. Here, we take task τ_2 as an example, which is released at time $t_2^0(0)$. As the CPU cores support the preemption, we only need to count the interference from the τ_0 and τ_1, which have higher priorities than task τ_2. Therefore, we need to find the maximum workload for τ_0 and τ_1 which works as interference to lower priority task τ_2. We first calculate the maximum workload of τ_0, given a time interval $t = t_2^0(1) - t_2^0(0)$. If τ_0 is at its 1st segment CL_0^0 at time $t_2^0(0)$, its workload for the time interval t is $W_0^0(t)$. Similarly, if τ_0 is at its jth segment CL_0^{j-1} at time $t_2^0(0)$, its workload for the time interval t is $W_0^{j-1}(t)$. As the τ_0 can be at its any segments, we have to use the worst-case workload $max(W_0^0(t), W_0^1(t), ...W_0^{M_0-1}(t))$ as the inference to task τ_2 during time interval t. By repeating the above process, we can get the worst-case workload from τ_1 $max(W_1^0(t), W_1^1(t), ...W_1^{M_1-1}(t))$ as the inference to task τ_2 during time interval t. If the worst case workload from τ_0 and τ_1 is less than the workload $N_{CPU} * t$ can be processed by N_{CPU} CPU cores, The minimal time $t_2^0(1)$ is when the task τ_2 has been executed by one unit. In this example, if $N_{CPU} = 1$ and $t_2^0(0) = 0$, then $t_2^0(1)$ is 23; if $N_{CPU} = 2$ and $t_2^0(0) = 0$, then $t_2^0(1)$ will be 3.

Lemma 4.3. *Given that the task τ_i has finished K units of workload in the $j+1th$ CPU segment CL_i^j at time $t_i^j(K)$, the $K+1$ units of workload in CL_i^j will be executed at least by time $t_i^j(K+1)$, where $t_i^j(K+1)$ is the minimal integer satisfying the following condition:*

$$\sum_{h\in hp(i)} max_{q\in[0\ M_h-1]}\{W_h^q\big(t_i^j(K+1)-t_i^j(K)\big)\} < N_{CPU} * \big(t_i^j(K+1)-t_i^j(K)\big), \tag{10}$$

where $hp(i)$ is the group of tasks that have a higher priority than task τ_i and M_h is the total number of CPU segments in task τ_h.

Proof. The computation provided by N_{CPU} CPU cores during $t_i^j(K+1) - t_i^j(K)$ is $N_{CPU} * (t_i^j(K+1) - t_i^j(K))$, which is the expression on the right of the inequality. Similar to Lemma 4.2, the run-time, worst-case (maximum) workload from task h can be upper bounded by picking the maximum workload starting from any CPU segment CL_h^j in task τ_h, i.e., $max_{q\in[0\ M_h-1]}\{W_h^q(t_i^j(K+1)-t_i^j(K))\}$. To calculate the time upper-bound when the task τ_i can finished 1 unit of workload after time $t_i^j(K)$, we only need to take into account the workload from the tasks with a higher priorities than i, which is noted by $hp(i)$ because the CPU segments access the CPU cores in a fixed-priority and preemptive manner. Therefore, the workload from higher-priority suspending tasks can be thereafter bounded above by the summation of $max_{q\in[0\ M_h-1]}\{W_h^q(t_i^j(K+1)-t_i^j(K))\}$. When the workload from higher-priority suspending tasks is less than the workload CPU can process from time $t_i^j(K))$ to time $t_i^j(K+1)$, the task τ_i has been executed for at least 1 unit. The minimal time $t_i^j(K+1)$ that satisfies (10) is the upper bound time of finishing $K+1$ units of workload in segment CL_i^j. ◻

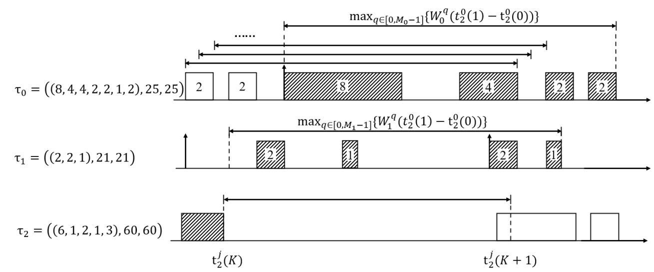


Figure 3: Example of the task τ_i that has been executed for 1 unit when τ_i is in the middle of a segment.

Case Study II A case study to calculate the time when the task τ_i has been executed for 1 unit when it is in the middle of a segment, is presented in Fig. 3. Here, we use the same taskset (in Case Study I) as an example. Given the τ_2 has finished K units of the $j+1th$ segment CL_2^j at time $t_2^0(K)$. Same to Case Study I, we need to find the maximum workload for τ_0 and τ_1 which works as interference to lower priority τ_2. The interference to task τ_2 from task τ_0 and τ_1 during a time interval t can be obtained by the worst-case workload $max(W_0^0(t), W_0^1(t), ...W_0^{M_0-1}(t))$ and $max(W_1^0(t), W_1^1(t), ...W_1^{M_1-1}(t))$. If the worst case workload from τ_0 and τ_1 is less than the workload $N_{CPU} * t$ can be processed by N_{CPU} CPU cores, The minimal time $t_2^j(K+1)$ is when the task τ_2 has been executed by one unit after it has finished K units of the $j+1th$ CPU segment. In this example, if $N_{CPU} = 1$, then $t_2^j(K+1)$ is $t_2^j(K) + 23$; if $N_{CPU} = 2$ then $t_2^j(K+1)$ is $t_2^j(K) + 3$.

Corollary 4.3.1. *Given the $j-1th$ CPU segment CL_i^j in task τ_i is ready at time $t_i^j(0)$, the worst-case response time CT_i^j of this CPU segment can be calculated by*

$$CT_i^j = t_i^j(CL_i^j) - t_i^j(0), \tag{11}$$

where $t_i^j(CL_i^j)$ is the time of finishing CL_i^j units of workload and it can be calculated with $t_i^j(0)$, $t_i^j(1)$, ..., $t_i^j(CL_i^j-1)$ following the Eq. 10 in Lemma 4.2.

Proof. $t_i^j(0)$ is the ready time of the the $j-1th$ CPU segment. This CPU segment has a length of CL_i^j units of workload. $t_i^j(CL_i^j)$ is the worst-case time of finishing CL_i^j units of workload. Therefore, the worst-case response time is the time interval from the segment is ready at $t_i^j(0)$ until CL_i^j units of workload in this task has been executed by time $t_i^j(CL_i^j)$. ◻

Corollary 4.3.2. *Given the $j-1th$ CPU segment CL_i^j in task τ_i is finished at time $t_i^j(CL_i^j)$, the next CPU segment and its ready time will be:*
If *$j = M_i - 1$, then the next CPU segment will be the first CPU segment CL_i^0 in the next period and its ready time will be the start of next period;*

Else $j < M_i - 1$, *then the next CPU segment will be* CL_i^{j+1} *and its ready time* $t_i^{j+1}(0)$ *will be* $t_i^{j+1}(0) = t_i^j(CL_i^j) + PT_i^j$.

Proof. This comes directly from the property of periodic real-time tasks and the task execution pattern in heterogeneous computing platforms modeled in Section 3. □

Theorem 4.4. *If the worst-case response time of task* τ_i *is less than its period* T_i*, then the worst-case response time of task* τ_i *is upper bounded by* R_i,

$$R_i = \sum_{j=0}^{M_i-1} CT_i^j + \sum_{j=0}^{M_i-2} PT_i^j, \tag{12}$$

where CT_i^j *and* PT_i^j *are derived in Eq.* (11) *and Eq.* (5)*, respectively.*

Proof. According to the task model for heterogeneous computing platforms, in the task, τ_i, the PE segment PL^j becomes ready immediately when its previous CPU segment CL_i^j finishes. And the CPU segment CL_i^j becomes ready immediately when its previous PE segment PL_i^{j-1} finishes. There is no delay between the CPU and PE segments. Therefore, the worst-case response time of task τ_i is upper bounded by the summation of the worst-case response time of each CPU and PE segment of task τ_i. □

The complete procedure of scheduling fixed-priority tasks on heterogeneous computing platforms can be described as follows: ① Grid search for a partitioning of PEs for each task based on federated scheduling and calculate the PE segment response time PT_i^j; ② The CPU segments are scheduled by fixed priority scheduling. The priority assignment is the optimal priority assignment detailed in Section 4.2; ③ If all the tasks can meet the deadline, then they are schedulable and otherwise go back to step ① to grid search for the next partitioning of PEs with federated scheduling. This schedulability test for hard deadline parallel GPU tasks can be summarized in Algorithm 1. Moreover, no additional assumptions or limitations are added to the computing platforms. The scheduling and response analysis can be directly applied to mainstream heterogeneous systems.

Algorithm 1: Scheduling and Response Time Analysis

Input: Number of CPU cores N_{CPU}, number of PEs N_{PE},
Task set τ with profiled parameters CL_i^j, PL_i^j, and P_i^j.
Output: Scheduability, PE allocation N_{PE_i}, Task priorities.
//Grid search for PE partitioning:
for N_{PE_1} = 1, ..., N_{PE} **do**
  for N_{PE_i} = 1, ..., $N_{PE} - \sum_{q=1}^{i-1} PE_q$ **do**
    for N_{PE_n} = 1, ..., $N_{PE} - \sum_{q=1}^{n-1} PE_q$ **do**
      //Calculate response times of PE segments:
      $PT_i^j = \frac{PL_i^j}{1-P_i^j+N_{PE_i}P_i^j}$;
      //Assign the priorities with AOPA in Section 4.2;
      //Calculate worst-case response time CT_i^j *= for every CPU segment* CL_i^j *by: Lemma 4.1, 4.2, and Corollary 4.2.1 with the recursive function:*
      $\sum_{h\in hp(i)} max_{q\in[0\ M(h)]}\{W_h^q(t_i^j(K+1)-t_i^j(K))\} < N_{CPU} * (t_i^j(K+1)-t_i^j(K))$;
      //Calculate worst-case end-to-end response time R_i *for each task using Theorem 4.3:*
      $R_i = \sum_{j=0}^{M_i-1} CT_i^j + \sum_{j=0}^{M_i-2} PT_i^j$;
      if $R_i \leqslant D_i$ *for all* $\tau_i \in \tau$ **then**
        Scheduability = 1;
        break out of all for loops;
return Schedulability;

4.2 Priority Assignment

In the proposed scheduling, the federated scheduling prevents the blockings and competition in accessing heterogeneous cores, and the fixed-priority scheduling supports the task preemption on the CPU side. Therefore, a key advantage of the proposed algorithm is that many essential properties of classic fixed-priority scheduling are kept as original. Meanwhile, in the derivation and analysis, we do not introduce any new constraints that conflict with classic Audsley's Optimal Priority Assignment Algorithm (AOPA) assumptions [39]. Audsley's AOPA is also effective in the proposed scheduling, and it will run in time $O(n^2)$ for n periodic tasks to find the optimal priority assignment.

4.3 Computational Complexity

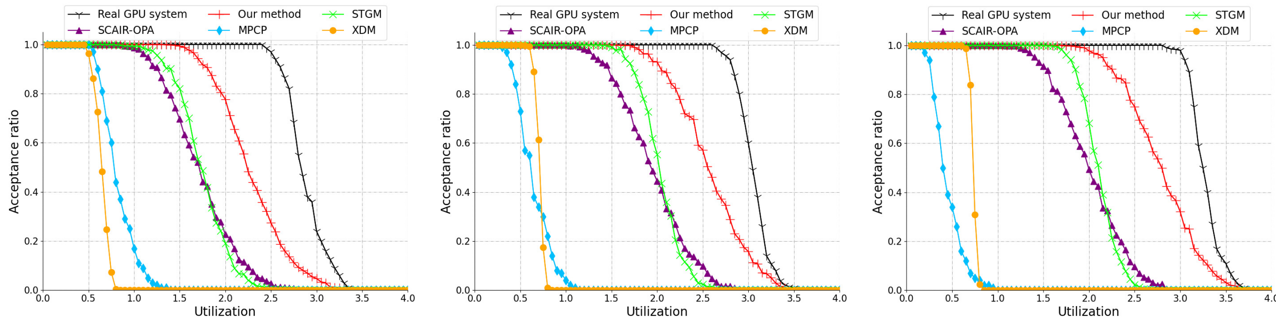
The proposed SHAPE scheduling and response time analysis contains the grid search on PEs spatial partitioning and fixed-priority scheduling on multi-core CPUs with priority assignments. Given the system and task models in Section 3, the grid search on PEs spatial partitioning has a complexity of $min(O({N_{PE}}^n), O(n^{N_{PE}}))$. The priority assignment has a complexity of $O(n^2)$ as discussed in above section. The analysis of fixed-priority tasks on multi-core CPUs has a complexity of $O(M_i^2)$. Therefore, the time complexity of the entire scheduling strategy with response time analysis is

$$min(O({N_{PE}}^n n^2 M_i^2), O(n^{N_{PE}+2} M_i^2)). \tag{13}$$

5 EVALUATION

5.1 Experimental Setup

In this section, extensive experiments are performed to evaluate the performance of the proposed scheduling approach with both numerical simulation and experiments on real CPU-GPU systems.
Scheduling Approaches: We compare the following state-of-the-



(a) Number of (CPU and PE) segments: 5 (b) Number of (CPU and PE) segments: 10 (c) Number of (CPU and PE) segments: 20

Figure 4: Schedulability in *"CPU and PE"*.

art scheduling algorithms, which can support partitioned PEs for general heterogeneous architectures.

(1) **XDM**: the pseudopolynomial-time analysis where we transform every task into a non-suspending task by modeling every suspension interval as computation segments and using the standard response time analysis under rate monotonic (RM) scheduling.
(2) **SCAIR-OPA** [16]: scheduling for self-suspension model under fixed priorities. This approach achieves great schedulability on the uni-core CPUs configuration.
(3) **Enhanced MPCP** [10]: real-time scheduling of hard deadline parallel tasks with a hybrid approach of the enhancements and practical insights for MPCP with self-suspension.
(4) **STGM** [11]: real-time GPU scheduling of hard deadline parallel tasks with partitioned PEs supported by the persistent threads. Busy-waiting scheduling and self-suspension scheduling are designed and analyzed.
(5) **SHAPE**: the proposed scheduling of fixed-priority tasks on heterogeneous architectures with multi CPU and many PEs.

System Implementation: The proposed scheduling is evaluated on a CPU-GPU heterogeneous computing platform with Intel i7-10700 CPU @ 2.90GHz and NVIDIA GTX 1660Ti GPU @ 1.50GHz. We implement the persistent threads to support the partitioning of PEs (i.e. Streaming Multiprocessors in GPUs). The persistent threads approach is a software workload assignment solution proposed to implement finer and more flexible PE-granularity GPU partitioning [32, 40, 41]. Specifically, each persistent threads block links multiple thread blocks of one PE segment and is assigned to one SM to execute for the entire hardware execution lifetime of the PE segment. We generate five types of tasks that have different features: 1) a computation task, consisting mainly of arithmetic operations; 2) a branch task containing a large number of conditional branch operations; 3) a memory task full of memory and register visits; 4) a special-function task with special mathematical functions, such as sine and cosine operations; and 5) a comprehensive task including all these arithmetic, branch, memory, and special mathematical operations. Each task performs floating-point operations on a vector which is determined by the PE segment length.

5.2 Unified Parallel Tasks

To compare the schedulability for different approaches, we measured the acceptance ratio with respect to a given goal for taskset utilization in each of five approaches and the real CPU-GPU system.

Table 1: Parameters for unified task generation

Parameters	Value
Number of tasks N in taskset	5
Task type	periodic tasks
Number of (CPU and PE) segments in each task	5, 10, 20
Number of tasksets in each experiment	1000
CPU segment length (ms)	[1 to 10]
Heterogeneous segment length (ms)	[1 to 10]
Task period and deadline	T_i/D_i
Number of CPU cores and PEs	2, 10
Priority assignment	AOPA/RMPA

We consider a heterogeneous computing platform with 2 CPU cores and 10 PEs and five parallel tasks run on this platform. We generated 1000 tasksets for each utilization level, with the following task configurations. The acceptance ratio of a level was the number of schedulable tasksets, divided by the number of tasksets for this level, i.e., 1000. We first generated a set of utilization rates, U_i, with a uniform distribution for the tasks in the taskset, and then normalized the tasks to the taskset utilization values for the given goal. For a complete comparison, we use both the methods in previous work [11] and [9, 16] to generate the CPU and PE segment lengths. We note the generation method in [11] as "*CPU and PE*". In "*CPU and PE*" we randomly generate both the CPU and GPU segment lengths, uniformly distributed within their ranges [1 10]. The deadline D_i of task i was set according to the generated segment lengths and its utilization rate: $D_i = \frac{\sum_{j=0}^{M_i-1} CL_i^j + \sum_{j=0}^{M_i-2} PL_i^j}{U_i}$. We note the generation method in [9, 16] as "*CPU Then PE*". In "*CPU Then PE*", the CPU segment lengths are uniformly distributed within their ranges [1 10]. The deadline D_i for task τ_i is determined by $D_i = \frac{\sum_{j=0}^{M_i-1} CL_i^j}{U_i}$. Then PE lengths of the tasks are generated according to a uniform random distribution, in one of three ranges depending on the PE length: $[0.01(D_i - \sum_{j=0}^{M_i-1} CL_i^j), 0.1(D_i - \sum_{j=0}^{M_i-1} CL_i^j)]$, $[0.1(D_i - \sum_{j=0}^{M_i-1} CL_i^j), 0.6(D_i - \sum_{j=0}^{M_i-1} CL_i^j)]$, and $[0.6(D_i - \sum_{j=0}^{M_i-1} CL_i^j), (D_i - \sum_{j=0}^{M_i-1} CL_i^j)]$. As multiple CPU cores and PEs are available (and used), the total utilization rate will be larger than 1. Task relative deadlines are implicit, and the period T_i is equal to the deadline D_i. The task priorities are determined with Audsley's Optimal Priority Assignment(AOPA) or rate monotonic priority assignment (RMPA). A summary of task generations is presented in Table 1.

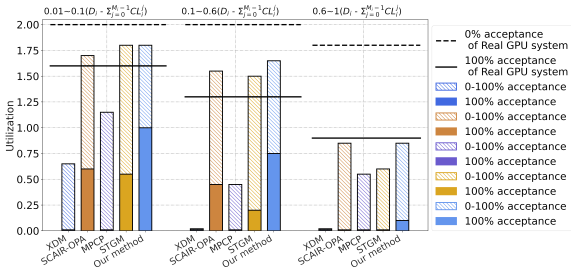


Figure 5: Number of (CPU and PE) segments: 5.

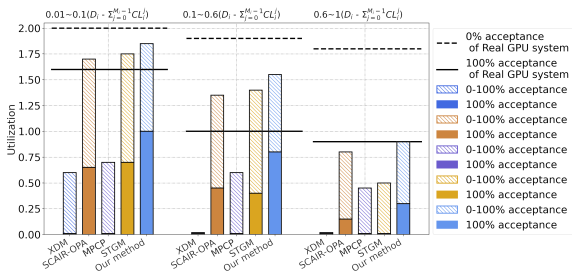


Figure 6: Number of (CPU and PE) segments: 10.

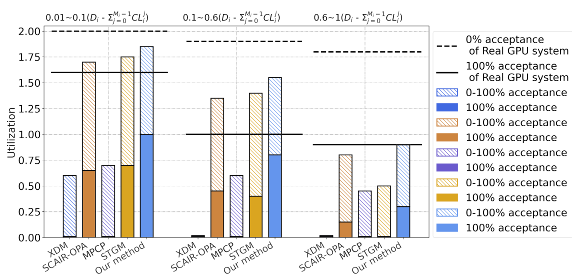


Figure 7: Number of (CPU and PE) segments: 20.

5.2.1 Schedulability in CPU and PE Task Generation. Following the task generation in "*CPU and PE*", Fig. 4 presents the acceptances under different resource utilization rates. We compare the response time analysis proposed in SHAPE, XDM, SCAIR-OPA, Enhanced MPCP, STGM, and also the schedulability presented by the CPU-GPU heterogeneous computing system with the proposed scheduling strategy. In Fig. 4(a), Fig. 4(b), and Fig. 4(c), the number of CPU and PE segments in each task are set to 5, 10, and 20. In the tests, the SCAIR-OPA and STGM improves the schedulability (i.e. the utilization rate at a given acceptance ratio) with XDM and MPCP because in the design of SCAIR-OPA and STGM, the partitioning of PEs is considered. STGM achieves a higher utilization at high acceptance ratio while SCAIR-OPA achieves a higher utilization at low acceptance ratio. The proposed SHAPE further improves the schedulability (the highest utilization rate at 100% acceptance ratio) by 73.3%, 18.5%, and 17.2% when there are 5, 10, and 20 CPU and PE segments.

To demonstrate the pessimism, we measured the area between the schedulability provided by the NVIDIA GPU systems and the proposed response time analysis in the related scheduling approaches. SHAPE reduces the pessimism by 47.8%, 54.4% 58.6%, compared with previous approaches when there are 5, 10, and 20 CPU and PE segments. Also it is notable that the pessimism, between the response time analysis in SHAPE and real GPU system, shrinks

Table 2: Parameters for versatile task generation

Tasks	Number of Segments	CPU Length	PE Length
Task 1 (AlexNet)	9CPU+8PE segments	[1 10]	[1 13]
Task 2 (VGG 11)	12CPU+11PE segments	[1 10]	[1 46]
Task 3 (VGG 19)	20CPU+19PE segments	[1 10]	[1 46]
Task 4 (GoogleNet)	23CPU+22PE segments	[1 10]	[1 36]
Task 5 (ResNet)	51CPU+50PE segments	[1 10]	[2 20]

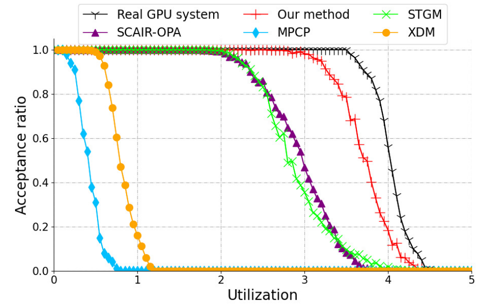


Figure 8: Schedulability for versatile tasks.

greatly at a lower acceptances. Meanwhile, as the number of segments increases, the schedulability in the real GPU system, proposed SHAPE, STGM, and SCAIR-OPA improve accordingly. This phenomenon matches the reported results in previous work [16, 31] on heterogeneous computing with uni-CPU and many PEs.

5.2.2 Schedulability in CPU then PE Task Generation. Similar to previous section, Fig. 5 to Fig. 7 present the highest utilization rates at 100% and 0% acceptance ratios, following the task generation in "*CPU then PE*". In Fig. 5, Fig. 6, and Fig. 7 the number of CPU and PE segments in each task are set to 5, 10, and 20, respectively. Across the experiments under different configurations, the naive approach XDM is hard to be effective for the 100% acceptance ratios. SCAIR-OPA, MPCP, and STGM improves the schedulability but still face the low utilization rate give the the 100% acceptance ratios. The proposed SAHPE achieves tremendous (11%.1 - 100%) and (0% - 18.8%) improvements of utilization rate when the acceptances are at 100% and reach 0%, compared with these related approaches.

In the experiments with 5 CPU and PE segments, the clusters are distinguished by the length of PE segments given the task period (deadline) and CPU segment lengths. For the case with short PE segments, (i.e. PE segments are generated by $[0.01(D_i - \sum_{j=0}^{M_i-1} CL_i^j)$, $0.1(D_i - \sum_{j=0}^{M_i-1} CL_i^j)])$, significant improvements are achieved by SHAPE over other approaches if the acceptance ratio is 100%. When the acceptance ratio reaches 0%, the improvements from SHAPE are not significant but still visible. Later, as the length of PE segments increases, the utilization rates begin to drop, especially for the acceptance is 100%. When the PE segments are generated by $[0.6(D_i - \sum_{j=0}^{M_i-1} CL_i^j), 1(D_i - \sum_{j=0}^{M_i-1} CL_i^j)]$, the utilization rates are close to 0 in previous approaches but the SHAPE can sill have a low utilization rate. Similar trends are observed in the experiments on 10 and 20 CPU and PE segments. Slight schedulability improvement is also found as the number of PE segments increase.

5.3 Versatile Parallel Tasks

The above experiments evaluate the scheduling performance on unified tasksets in which the tasks have the same topology like length distribution and the number of subtasks. This section will test the scheduling algorithm under versatile parallels tasks. We generate the tasks based on classic convolutional neural network (CNN) topology: AlexNet, VGG 11, VGG 19, GoogleNet, and ResNet. The number of PE segments is based on the layers of the network. The segment length is also randomly generated between the smallest and largest length of each segment, which are calculated based on the smallest and largest layer of the network for that task. Similarly, the deadline D_i of task i was set according to the generated segment lengths and its utilization rate: $D_i = \frac{\sum_{j=0}^{M_i-1} CL_i^j + \sum_{j=0}^{M_i-2} PL_i^j}{U_i}$. A summary of versatile task generations is presented in Table 2. Fig. 8 presents the schedulability of different approaches and the real GPU systems, noted by the utilization rates and corresponding acceptance ratio. The proposed SHAPE achieves 27.8% utilization improvements at the same acceptance ratio compared with previous approaches. Meanwhile, the pessimism between the response time analysis and real GPU system is further reduced by 70.9%.

6 CONCLUSION

Targeting heterogeneous architectures with multiple preemptive CPUs and many non-preemptive PEs, we proposed scheduling strategy and response time analysis, SHAPE. It achieves up to 100% and 27.8% schedulability improvement on unified and versatile machine learning tasks, and the pessimism is further reduced by 70.9%. The essential properties in SHAPE enable it to collaborate with the classic optimal priority assignment algorithm. Since our method is developed from general heterogeneous architectures, it can be directly applied to the off-the-shelf heterogeneous computing platforms, such as CPU-GPU and CPU-FPGA systems. In this paper, the proposed scheduling and response time analysis are built on the platforms where PEs share the same architecture. We will work on scheduling of the computing platforms integrating multiple types of heterogeneous PEs is our future work.

REFERENCES

[1] Jeff Anderson, Armin Mehrabian, Jiaxin Peng, and Tarek A El-Ghazawi. Extreme heterogeneity in deep learning architectures., 2019.

[2] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[3] Philipp Michel, Joel Chestnutt, Satoshi Kagami, Koichi Nishiwaki, James Kuffner, and Takeo Kanade. Gpu-accelerated real-time 3d tracking for humanoid locomotion and stair climbing. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 463–469. IEEE, 2007.

[4] Jack Choquette and Wish Gandhi. Nvidia a100 gpu: Performance & innovation for gpu computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43. IEEE Computer Society, 2020.

[5] Steve Leibson and Nick Mehta. Xilinx ultrascale: The next-generation architecture for your next-generation architecture. *Xilinx White Paper WP435*, 143, 2013.

[6] Benjamin Schwaller, Barath Ramesh, and Alan D George. Investigating ti keystone ii and quad-core arm cortex-a53 architectures for on-board space processing. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2017.

[7] Ronald B Brightwell. Resource management challenges in the era of extreme heterogeneity. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2018.

[8] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, et al. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1):144–207, 2019.

[9] Wen-Hung Huang and Jian-Jia Chen. Self-suspension real-time tasks under fixed-relative-deadline fixed-priority scheduling. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1078–1083. IEEE, 2016.

[10] Pratyush Patel, Iljoo Baek, Hyoseung Kim, and Ragunathan Rajkumar. Analytical enhancements and practical insights for mpcp with self-suspensions. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 177–189. IEEE, 2018.

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

[12] R Gandham, Y Zhang, K Esler, and V Natoli. Improving gpu throughput of reservoir simulations using nvidia mps and mig. In *Fifth EAGE Workshop on High Performance Computing for Upstream*, volume 2021, pages 1–5. European Association of Geoscientists & Engineers, 2021.

[13] Nathan Otterness and James H Anderson. Exploring amd gpu scheduling details by experimenting with "worst practices",. In *Proceedings of the 29th International Conference on Real-Time Networks and Systems*, 2021.

[14] Nathan Otterness and James H Anderson. Amd gpus as an alternative to nvidia for supporting real-time workloads. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[15] Konstantinos Bletsas, Neil Audsley, Wen-Hung Huang, Jian-Jia Chen, and Geoffrey Nelissen. Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions. Technical report, CISTER-Research Centre in Realtime and Embedded Computing Systems, 2015.

[16] Wen-Hung Huang and Jian-Jia Chen. Schedulability and priority assignment for multi-segment self-suspending real-time tasks under fixed-priority scheduling. In *Technical report*. Technical University of Dortmund, 2015.

[17] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[18] Glenn A Elliott and James H Anderson. Globally scheduled real-time multiprocessor systems with gpus. *Real-Time Systems*, 48(1):34–74, 2012.

[19] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE, 2013.

[20] Vladislav Golyanik, Mitra Nasri, and Didier Stricker. Towards scheduling hard real-time image processing tasks on a single gpu. In *International Conference on Image Processing (ICIP)*. IEEE, 2017.

[21] Husheng Zhou, Soroush Bateni, and Cong Liu. Sˆ 3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 190–201. IEEE, 2018.

[22] Enrico Rossi, Marvin Damschen, Lars Bauer, Giorgio Buttazzo, and Jörg Henkel. Preemption of the partial reconfiguration process to enable real-time computing with fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(2):1–24, 2018.

[23] Houssam-Eddine Zahaf, Giuseppe Lipari, Smail Niar, et al. Preemption-aware allocation, deadline assignment for conditional dags on partitioned edf. In *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 1–10. IEEE, 2020.

[24] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. *ACM SIGARCH Computer Architecture News*, 43(1):593–606, 2015.

[25] Can Basaran and Kyoung-Don Kang. Supporting preemptive task executions and memory copies in gpgpus. In *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. IEEE, 2012.

[26] Ivan Tanasic, Isaac Gelado, Javier Cabezas, Alex Ramirez, Nacho Navarro, and Mateo Valero. Enabling preemptive multiprogramming on gpus. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 193–204. IEEE, 2014.

[27] Husheng Zhou, Guangmo Tong, and Cong Liu. Gpes: A preemptive execution system for gpgpu computing. In *Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015.

[28] Guoyang Chen, Yue Zhao, Xipeng Shen, and Huiyang Zhou. Effisha: A software framework for enabling effficient preemptive scheduling of gpu. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.

[29] Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233. IEEE, 2019.

[30] Jounghoo Lee, Jinwoo Choi, Jaeyeon Kim, Jinho Lee, and Youngsok Kim. Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 247–252. IEEE, 2021.

[31] An Zou, Jing Li, Christopher D Gill, and Xuan Zhang. Rtgpu: Real-time gpu scheduling of hard deadline parallel tasks with fine-grain utilization. *arXiv preprint arXiv:2101.10463*, 2021.

[32] Chao Yu, Yuebin Bai, Hailong Yang, Kun Cheng, Yuhao Gu, Zhongzhi Luan, and Depei Qian. Smguard: A flexible and fine-grained resource management framework for gpus. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[33] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 754–768, 2019.

[34] Shuai Che, Jeremy W Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–11, 2011.

[35] Raphael Landaverde, Tiansheng Zhang, Ayse K Coskun, and Martin Herbordt. An investigation of unified memory access performance in cuda. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2014.

[36] William F Gilreath and Phillip A Laplante. Parallel architectures. In *Computer Architecture: A Minimalist Perspective*, pages 113–132. Springer, 2003.

[37] Adam Betts and Alastair Donaldson. Estimating the wcet of gpu-accelerated applications using hybrid analysis. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 193–202. IEEE, 2013.

[38] Tao Chen, Alexander Rucker, and G Edward Suh. Execution time prediction for energy-efficient hardware accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 457–469, 2015.

[39] Neil C Audsley. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times.* Citeseer, 1991.

[40] Kshitij Gupta, Jeff A Stuart, and John D Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing-Foundations & Applications of GPU, Manycore, and Heterogeneous Systems (INPAR 2012)*. IEEE, 2012.

[41] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. Enabling and exploiting flexible task assignment on gpu through sm-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 2015.