Offloading Accelerator-intensive Workloads in CPU-GPU Heterogeneous Processors

Nandinbaatar Tsog*, Saad Mubeen*, Fredrik Bruhn*†, Moris Behnam*, and Mikael Sjödin*
*School of Innovation, Design and Engineering
Mälardalen University, Box 883, 721 23 Västerås, Sweden
Email: firstname.lastname@mdh.se
‡Unibap AB (publ), Svartbäcksgatan 5, 753 20 Uppsala, Sweden
Email: f@unibap.com

Abstract—**Autonomous vehicular systems require computer vision and intelligent on-board decision making functionalities that include a mix of sequential and parallel workloads. The execution times of the workloads and power consumption in these functionalities can be lowered by utilizing the accelerators (e.g., GPU) instead of running the workloads entirely on the host processing units (CPU). However, allocating all the parallelizable workload to accelerators can create a computation bottleneck in the accelerators that, in turn, can have an adverse effect on schedulability of the systems. This paper presents a novel framework that can allocate the accelerate-intensive workloads to the accelerators as well as to the non-accelerated host processing units. Within the context of this framework, the paper introduces five offloading techniques to mitigate the accelerator-intensive workloads by utilizing excess capacity of non-accelerated processing units under dynamic scheduling in CPU-GPU heterogeneous processors. The proposed techniques are evaluated using simulation experiments. The evaluation results indicate that one of the proposed techniques can achieve up to 16% improvement in schedulability of the task sets compared to the traditional non-offloading technique.**

I. Introduction

Employing accelerators in modern embedded platforms increases the diversification of embedded system applications. Examples of the accelerators include graphics processing units (GPUs), field-programmable gate arrays (FPGAs) and digital signal processors (DSPs), which are often used in parallel programming applications like computer vision. The accelerators often perform better than general-purpose processors (i.e., central processing units (CPUs)) with respect to latency and power consumption. For example, in autonomous vehicles, computer vision and intelligent decision making often require the use of heterogeneous processors (e.g., CPU and GPU) [1], [2]. However, the accelerators act as shared resources within heterogeneous processors [3], [4], which brings challenges of synchronization and blocking.

While the processing trend has been shifting from single-core to multi- and many-core processors as well as to heterogeneous processors, the development of single-core processors is also progressing, e.g., AMD Ryzen 5000 series with Zen3 architecture CPUs[1] and 11-th Generation Intel Core i9 processors[2]. In other words, the host processing units, CPUs, in heterogeneous processors are becoming more and more capable of assisting the accelerators in computing accelerator-intensive workloads. This paper focuses on techniques to offload accelerator-intensive workloads from GPUs to non-accelerated host processing units in the systems with heterogeneous processors. We assume that accelerator-intensive workloads contain a segment, a parallel segment, which can be parallelizable on the accelerators. Furthermore, in this paper, we consider that the applications that run on the heterogeneous processors are constrained by real-time requirements.

The main contributions in this paper are as follows:

- We propose a new framework to allocate accelerator-intensive workloads in CPU-GPU heterogeneous processors. The proposed framework utilizes the alternative executions of parallel segments of the workloads [4], [5] and the server-based scheduling [6]–[8].
- Based on the proposed framework, we introduce five techniques for mitigating the accelerator-intensive workloads by lowering the overuse of the accelerators. The proposed techniques use the resource-reservation mechanism by means of servers to offload the execution of parallel segments of the workload to non-accelerated host processing units.
- We perform a comparative evaluation of the proposed offloading techniques with respect to the baseline technique that always executes the parallel segments on the accelerators. The evaluation is performed on the basis of schedulability of the task sets and the time to perform the offloading.

The rest of the paper is organized as follows. Section II discusses the related work. Section III presents the proposed system model, followed by the proposed workload allocation framework in Section IV. The offloading techniques for mitigating the accelerator-intensive workloads are presented in Section V. Section VI presents the experimental evaluation. Finally, Section VII discusses the conclusion and future work.

II. Related Work

Historically, the adoption of heterogeneous processors is intimately bound to the development of supercomputers, especially, in the area of distributed heterogeneous supercomputing [9]. The execution times of the workloads can vary a lot

[1] https://www.amd.com/en/processors/ryzen

[2] https://www.intel.com/content/www/us/en/products/details/processors/core/i9.html

978-1-7281-2989-1/21/$31.00 ©2021 IEEE

depending on what type of processing units they are executed on. To this end, there are several existing works that focus on how to allocate applications to the appropriate processing units in order to achieve the best-case execution time, i.e., the shortest execution time [5], [10], [11]. In contrast, the work presented in this paper focuses on offloading the accelerator-intensive workloads, constrained by real-time requirements, e.g., deadlines on the response times of the workloads, to the available non-accelerated host processing units.

The heterogeneous processors considered in this paper consist of mainly two parts: (i) a host processing unit, CPU, and (ii) accelerator(s) that include GPUs and FPGAs, among others. There exist several research trends on how to tackle heterogeneous processors in real-time systems. One of the research trends is to explore the properties of accelerators in heterogeneous processors since a host processing unit is a well-studied single-core CPU. The existing works in this regard include TimeGraph [12], Gdev [13], the black-box method [14], to mention a few.

Another line of existing works targets resource management in the systems that use heterogeneous processing units. There are several works [15]–[17] that focus on splitting a task on accelerators for improving the schedulability. Moreover, TimeGraph [12], GPUSync [18], and the works by Kim et al. [19] and Biondi et al. [20] consider schedulability analysis of the systems that use heterogeneous processors. These works focus on accelerators, which obviously offer better (shorter) execution times of the compute-intensive workloads compared to the executions on the host processing units. On the other hand, the work in this paper aims at mitigating the accelerator-intensive workload by efficiently offloading it to the non-accelerated host possessing units.

There exist several works that support server-based scheduling on single- and multi-core CPU(s) such as the constant bandwidth server (CBS) [6], total bandwidth server (TBS) [7], polling server (PS), sporadic server (SS) and deferrable server (DS) [8]. The work presented in this paper uses the DS. Some of the existing works also address the challenge of using the server-based scheduling in accelerators. For instance, the works in [21], [22] show that the server-based scheduling on accelerator(s) can improve the schedulability of the systems that use heterogeneous processors. In comparison to these works, the work presented in this paper uses the server-based scheduling in the host processing units instead of accelerators. The rationale behind this decision is that the proposed framework offloads the accelerator-intensive workloads to host processing units for efficiently utilizing their excess resources to assist the accelerators.

The idea of using alternative executions of parallel segments of real-time workloads is discussed in a few works [4], [23]. Baruah [23] applies conditional branching by using the if-then-else construct for two or more alternative executions of a workload. Moreover, a scheduling approach is introduced based on the conditional DAG model for reserving the necessary amount of computing resources. Tsog et al. [4] discuss a static allocation of real-time tasks using alternative execution of parallel segments of the tasks. Both works construct the fundamental of alternative executions of segments under real-time constraints. However, dynamic allocation of tasks using the alternative executions of parallel segments is missing from the state of the art. Provisioning of such an allocation is the main focus of the work presented in this paper.

III. System Model

We consider compute-intensive tasks that heavily require the use of accelerators such as GPUs. A periodic task, τ_i, is characterized by the tuple $\{S_i, D_i, T_i\}$, where S_i represents the set of finite sequence of execution segments of the task, D_i identifies the relative deadline of the task, and T_i represents the task's period. Furthermore, S_i consists of l sequential and parallel segments, $\{S_{i,1}, \ldots, S_{i,l}\}$, where $l \in \mathbb{N}$, i.e., it follows the traditional fork-join task model [24]. Regarding a parallel segment ($S_{i,j}, 1 < j < l$), we consider the model of alternative execution of parallel segments according to the work in [4]. Fig. 1 illustrates these execution segments as well as alternative executions of parallel segments. Traditionally, a sequential segment is executed on CPU as it can only be executed in sequential manner. In contrast, a parallel segment can be executed either in a sequential/parallel manner.

In most cases, executing a parallel segment in parallel manner improves its execution time compared to executing it in a sequential manner. Hence, the developers tend to allocate parallel segments to GPU (or on CPU with multi-threading techniques) to execute them in parallel manner as shown in Fig. 1(a). This may not be efficient in all cases, especially when a parallel segment can be executed in a sequential manner if the GPU is busy serving other parallel segments. Fig. 1(b) illustrates the two alternatives to execute the parallel segments. In this paper, we consider that a parallel segment is executed in parallel manner only on GPU and in sequential manner only on CPU. This means that an instance of a parallel segment can be executed on GPU in parallel manner, while another instance of the same parallel segment can be executed on CPU in sequential manner.

The execution time $C_{i,j}$ of any segment $S_{i,j}$ of task τ_i is described by (1), where $h_{i,j}$ expresses the allocation decision of the segment $S_{i,j}$ to the processing units, and $C_{i,j}^{\texttt{CPU}}$ and $C_{i,j}^{\texttt{GPU}}$ are the execution times of the segment $S_{i,j}$ on CPU and GPU, respectively. In other words, $h_{i,j} = \texttt{CPU}$ and $h_{i,j} = \texttt{GPU}$ mean that the segment is allocated to CPU and GPU, respectively.

$$C_{i,j} = \begin{cases} C_{i,j}^{\texttt{CPU}} & \text{, if } h_{i,j} = \texttt{CPU} \\ C_{i,j}^{\texttt{GPU}} & \text{, if } h_{i,j} = \texttt{GPU} \end{cases} \tag{1}$$

We consider that the execution time C_i of task τ_i is the summation of the total execution times of its sequential segments on CPU and parallel segments on GPU, i.e.,

$$C_i = \sum_{\forall \tau_{i,j} | h_{i,j} = \texttt{CPU}} C_{i,j} + \sum_{\forall \tau_{i,j} | h_{i,j} = \texttt{GPU}} C_{i,j} \tag{2}$$

We define $C_i^{\texttt{CPU}}$ and $C_i^{\texttt{GPU}}$ as the total execution times of task τ_i on CPU and GPU respectively.

$$C_i^{\texttt{CPU}} = \sum_{\forall \tau_{i,j} | h_{i,j} = \texttt{CPU}} C_{i,j} \tag{3}$$

$$C_i^{\texttt{GPU}} = \sum_{\forall \tau_{i,j} | h_{i,j} = \texttt{GPU}} C_{i,j} \tag{4}$$

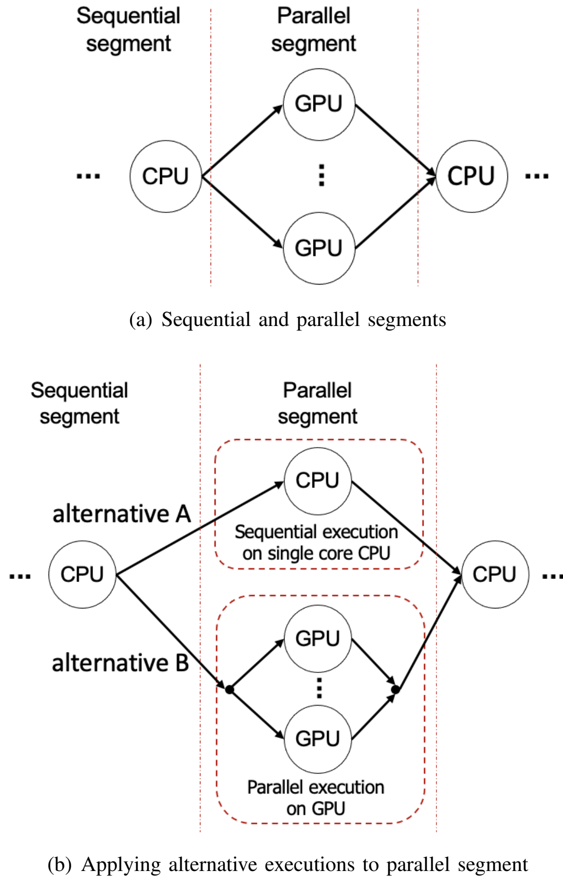

Fig. 1: Difference between parallel segments applied with and without alternative executions.

Hence, the execution time C_i can be represented as:

$$C_i = C_i^{\text{CPU}} + C_i^{\text{GPU}} \tag{5}$$

Intuitively, the utilization of task τ_i is defined as:

$$U_i = (C_i^{\text{CPU}} + C_i^{\text{GPU}})/T_i \tag{6}$$

Note that the value of C_i depends upon how its segments are allocated. To distinguish if a task τ_i is CPU-heavy or GPU-heavy, we consider a metric, μ_i, which is defined as conversion ratio between C_i^{CPU} and C_i^{GPU}, i.e,

$$\mu_i = C_i^{\text{CPU}}/C_i^{\text{GPU}} \tag{7}$$

The objective of this paper is to lower the utilization of highly-utilized accelerators and reduce the response times of middle and low priority tasks. The response time of task τ_i is expressed by the parameter R_i.

IV. Proposed Workload Allocation Framework

The proposed allocation framework, illustrated in Fig. 2, considers the allocation of parallel and sequential segments of tasks differently. A given sequential segment is allocated to a CPU, while different sequential segments might be allocated to different CPUs, and these allocations are fixed. This means, a sequential segment that is ready to execute will be allocated to the appropriate CPU queue. The CPU queues follow the priority-based preemptive scheduling policy. The priorities are assigned according to the rate-monotonic algorithm. However, any other scheduling policy can be used in the CPU queues using the proposed allocation framework, e.g., earliest deadline first.

We introduce an allocator to efficiently manage the execution of parallel segments. All parallel segments that are ready to execute are placed in the allocator queue as shown in Fig. 2. The allocator, in turn, decides which parallel segment to allocate to the GPU or CPU depending upon the availability of these compute units. As the first step, we consider the priority-based arbitration in the allocator queue such that the highest priority parallel segment is selected for allocation to the GPU or CPU. Other arbitration policies can also be applied such as the first-come-first-served policy. However, incorporation of the other policies within the proposed framework is left for the future work.

We assume that only one parallel segment executes on the GPU at a time and it is non-preemptive. This assumption is in line with the assumption of running one task on GPU at a time considered in the previous works [18], [19]. Furthermore, the policies for scheduling of tasks in the GPU proposed by Elliot et al. [18] and Kim et al. [19] can be used. The main difference between these policies is whether to keep the selected ready parallel segment in busy waiting or self suspension if the compute resources are busy. In this paper, we consider the self-suspension policy [19].

The proposed framework relies on the server-based dynamic scheduling to ensure that CPU resources are reserved to execute the parallel segments. For this purpose, we adopt the deferrable server (DS) with synchronization to m multi-core CPUs [8]. Other servers such as the Constant Bandwidth Server, Total Bandwidth Server, and Sporadic Server are also applicable. The DS, considered in this paper, is a set of m synchronized deferrable servers (SDS), which is characterized by an $(m+1)$-tuple $\{T_s, Q_{s_1}, Q_{s_2}, ..., Q_{s_m}\}$, where T_s is the common replenishment period of the SDSs and Q_{s_i} is the maximum capacity/budget of the i-th DS. We assume that each server has the highest priority in its respective CPU queue. Parallel segments are allocated to a single CPU server at a time. However, this restriction does not limit the execution of a parallel segment on different CPU servers. Thus, it is possible to dynamically allocate a given parallel segment to different servers throughout its execution.

V. Offloading Techniques

Traditionally, all parallel segments are allocated to accelerators in order to exploit the high-performance computing potential of the accelerators. However, this can have an adverse effect on the response times of the middle- and lower-priority tasks allocated to the accelerators when priority-based arbitration is used in the allocator queue. To address this, we consider to offload accelerator-intensive workloads, especially the middle and low priority parallel segments, to non-accelerated host processing units. However, the execution time of the parallel segments on non-accelerated devices is mostly longer than the execution time on the accelerator. That is, μ_i (see Equation 7) can reach up to 20 [25], [26] or even

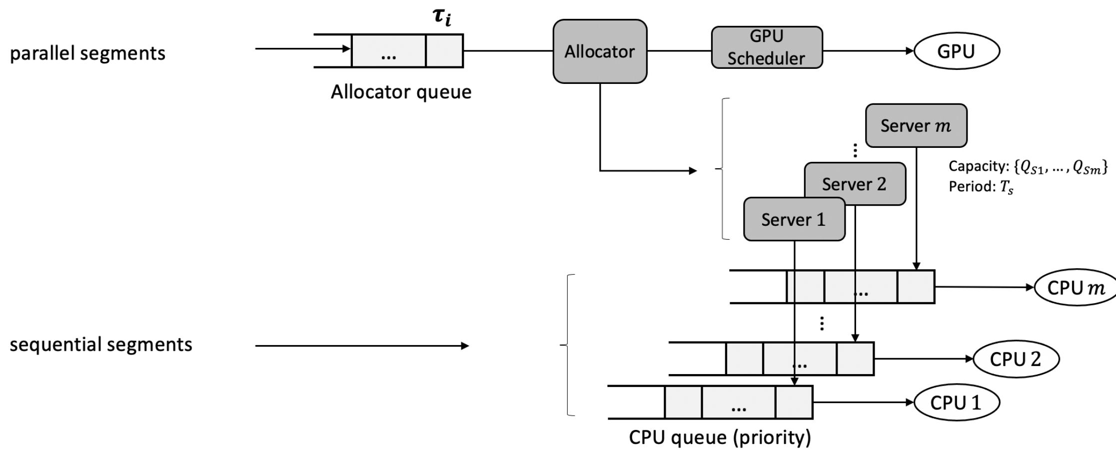

Fig. 2: Proposed workload allocation framework based on server-based global scheduling of heterogeneous processors.

more as it is related to the type of CPU and GPU. In this paper, we consider the value of μ_i to be between 1.5 and 10. Note that due to the offloading strategy, the execution of parallel segments on non-accelerated processing units using servers can block the sequential segments of higher priority tasks.

There are two main concerns regarding the offloading techniques: (i) how to lower the overhead of allocation time while dynamically allocating the parallel segments to the compute units using the offloading techniques? and (ii) how to mitigate the impact of the allocation techniques on higher priority parallel segments? That is, how to reduce the time between the instant when the task is ready and the instant when it starts executing in the GPU? We introduce five offloading heuristic techniques for dynamic allocation of accelerator-intensive tasks in Sections V-B- V-F in order to explore the potential impacts using these techniques. Optimal algorithms for allocating tasks to heterogeneous processing are NP-hard problems [27]. The main difference among these techniques is how they address and balance the above mentioned concerns. Note that we do not consider the genetic algorithms based techniques as their computation time is very high due to which they are not preferable for dynamic allocations [4], [28].

A. Baseline: Default Allocation Technique (DAT)

The DAT allocates all parallel segments to the GPU. Hence, this technique does not offload parallel segments to the non-accelerated host processing units. We consider the DAT as the baseline for comparative evaluations (discussed in the next section).

B. Naive Offloading Technique (NOT)

The NOT has no intelligent decision mechanism for offloading the parallel segments. This technique is intended to reveal the difference of computing performance between the host processing unit and accelerator devices. In short, this technique allocates parallel segments to the devices in the order of their available computing capacity. This technique can be expressed in the following steps.

- Step 1. Compute a list of available devices based on their computing capacity.
- Step 2. Allocate the parallel segment with the highest priority in the allocator queue to the device with the highest computing capacity in the list of available devices.
- Step 3. Repeat Step 2 until there are no available devices or no tasks in the allocator queue.

C. Min-min Fashioned Offloading Technique (MOT)

The min-min bin-packing approach is a well-known approach by packing rule and packing results [28]. The MOT offloading technique for parallel segments to non-accelerated devices is based on the min-min bin-packing approach. This technique is described by the following steps.

- Step 1. When there are available accelerators, allocate the highest priority parallel segments from the allocator queue to the accelerators.
- Step 2. If the allocator queue is empty then either all the allocator segments have been allocated to the accelerators and/or there is no ready parallel segment. If the allocator queue is not empty and there are no more accelerators available then go to next step.
- Step 3. Select the parallel segment with the highest priority in the allocator queue.
- Step 4. Calculate the summation of the current waiting time of the selected segment in the allocator queue and its execution time on the accelerator (GPU).
- Step 5. Calculate the difference between the summation (calculated in Step 4) and the execution time of the parallel segment on the CPU. If the difference is negative, allocate the segment to the CPU. Otherwise, allocate the segment to the GPU.
- Step 6. Repeat Step 1 if the allocator is not empty.

D. Speedup Classifier Based Technique (SCT)

The offloading technique is based on the speedup classifier bin-packing algorithm, which is studied in several works within the context of heterogeneous processors [4], [5]. As illustrated in Fig. 3, the different versions of this technique can exist based on selection of the classifier. In this paper,

we select μ_i as the classifier. The SCT follows the following steps.

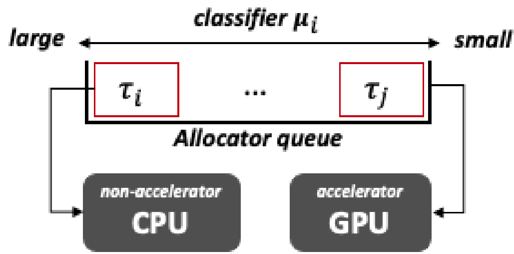

Fig. 3: A sorted task queue based on speedup classifier.

- Step 1. Sort the parallel segments in the allocator queue according to the speedup classifier, which is μ_i in this case. This means, the segments with smaller μ_i will be stored on right side (close to the accelerator) and the segments with larger μ_i will be at placed at the left side (close to the non-accelerated device).
- Step 2. Allocate the parallel segments with the smallest μ_i to an accelerator with the higher computing capacity when there are available accelerator(s). Repeat this step until there is no available accelerator.
- Step 3. If there are parallel segments in the allocator queue and non-accelerated devices (CPUs) are available, allocate the parallel segment with the highest μ_i to the available CPU. Repeat this step until all CPUs are unavailable.

A disadvantage of this technique is that the parallel task segments that have the smallest and largest values of the μ_i classifier are prioritized compared to those with average values of the μ_i classifier. This means that a higher priority parallel segment with the average value of the μ_i classifier can be blocked by a lower priority parallel segment with a higher or lower value of the μ_i classifier.

E. Synchronized Servers Technique (SST)

This technique uses several servers to offload parallel segments to the non-accelerated devices. This technique can be described by the following steps.

- Step 1. Allocate the highest priority parallel segments to the available accelerators.
- Step 2. Repeat Step 1 until no more available accelerators.
- Step 3. If no parallel segment is currently using any server capacity, pick the parallel segment with the highest priority in the allocator queue. Otherwise, jump to Step 6.
- Step 4. Calculate the summation of the current waiting time of the selected segment in the allocator queue and its execution time on the accelerator (GPU).
- Step 5. Calculate the difference between the summation (calculated in Step 4) and the execution time of the parallel segment on the CPU. If the difference is negative, allocate the segment to the CPU. Otherwise, allocate the segment to the GPU. Select a parallel segment with the second-highest priority in the allocator queue and repeat Step 4.
- Step 6. In step 3, if there is a parallel segment that is currently using a server budget, then check whether the current server has a left over capacity to serve new requests. If yes, continue to run the parallel segment on the same server. Otherwise, switch the parallel segment to the next available server.
- Step 7. Repeat Step 6 until no more servers are available or the parallel segment completes its execution. In the former case, the execution of the parallel segment should be postponed until the start of the next period of the server and then repeat Step 6.

F. Efficient Offloading Technique (EOT)

We adapt the previous techniques to propose an efficient offloading technique. The EOT consists of the following steps.

- Steps 1-2. The first two steps are the same as that of Steps 1-2 in the SST.
- Step 3. If no parallel segment is currently using any server capacity, pick the parallel segment of the lowest priority task in the allocator queue as the lowest priority task tends to miss its deadline. Otherwise, jump to Step 5.
- Step 4. Allocate the selected parallel segment to a server, which runs on next CPU to the CPU that handles the sequential segment of the same task. Allocate the selected parallel segment to a server with the index of $i+1$ when the index of the server that serves the sequential segment of the same task is i. It is worth to note that we consider the server index of 1 instead of $i+1$ when i equals to m. This avoids to block the higher-priority sequential segments of other tasks assigned to the same server of the sequential segments of the select parallel segment's task.
- Step 5. In step 3, if there is a parallel segment that is currently using a server budget, then check whether the current server has a left over capacity to serve new requests. If yes, continue to run the parallel segment on the same server. Otherwise, switch the parallel segment to the next available server.
- Step 6. Repeat Step 5 until no more servers are available or the parallel segment completes its execution. In the former case, the execution of the parallel segment should be postponed until the start of the next period of the server and then repeat Step 5.

VI. Experimental Evaluation

A number of synthetic experiments are performed to evaluate a wide range of application parameters using the proposed framework and offloading techniques.

A. Task Set Generation and Experimental Setup

Table I illustrates the configuration that is used to generate the task sets. The task generation technique is based on the UUniFast algorithm [29]. The UUniFast is used in two ways to generate a task. First, using the given system utilization U, the UUniFast generates n random task utilizations for n tasks. For example, U_i represents the utilization for the task τ_i. The simulator initializes the following basic parameters of a task:

- period T_i,
- number of parallel segments
- ratio of the length of parallel and sequential segments, and
- conversion ratio of parallel segments μ_i.

The UUniFast generates the random length of parallel and sequential segments based on each task utilization and the total

execution times of parallel and sequential segments. The total execution times of parallel and sequential segments are derived from the task utilization, the period and the ratio of the length of parallel and sequential segments. In order to generate an alternative of parallel segments (i.e., the sequential execution of parallel segments), the conversion ratio of parallel segments μ_i is used. The value of μ_i is randomly selected between 1.5 and 10, which is in line with the existing experimental studies [25], [30], [31].

TABLE I: Initial configuration of task set generation.

Parameters	Values
Number of CPU cores (N_p)	4, 8
Number of GPUs	1
Number of tasks (n)	$[N_p, 10N_p]$
System utilization (U)	0.5-2
Task period and deadline ($T_i = D_i$)	[30, 500]ms
Ratio of parallel to sequential segments length (C_i^{GPU}/C_i^{CPU})	[0.1, 3]
Number of parallel segments per task (η_i)	1-3
Conversion ratio of parallel segments (μ_i)	[1.5-10]
Server utilization (Q_{s_i}/T_s)	[20-40]%
The common period of the SDSs (T_s)	[30, 500]ms

In each experiment, the synthetic experiment simulator is run until the schedulability of the task set converges to the given condition. The simulator is executed minimum 100 times with 3,000,000 cycles to get the variance of schedulability of the task sets. Based on the variance value, the simulator continues to execute until 200 times in fast-converging cases and 500 times in slow-converging cases.

B. Offloading Techniques

We perform comparative evaluation of five offloading techniques (NOT, MOT, SCT, SST and EOT), presented in Section V, with respect to the DAT technique (Section V). The DAT acts as the baseline technique as it does not support offloading the parallel segments to non-accelerated processing units. Note that the NOT, MOT, SCT are focused on executing parallel segments on a CPU server, while the SST and EOT consider to execute parallel segments on multiple synchronized CPU servers. The input to the offloading techniques is a set of tasks that must fulfill the input requirements of the task model (Section III) such as segments' deadlines. The execution on CPU is preemptible and self-suspending, while the execution on GPU is non-preemptive. The execution traces in the experiments are evaluated until the hyperperiod (least common multiple of all periods) of the task set, which takes 3,000,000 cycles.

C. Evaluation Results

This subsection presents the evaluation results of the following three groups of experiments.

- *Experiment A* focuses on the comparative evaluation of the offloading techniques.

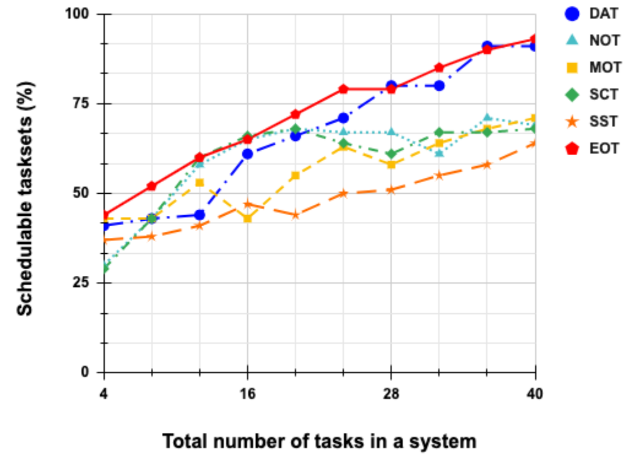

Fig. 4: Schedulable task sets w.r.t. the total number of tasks in the system with 4 CPU cores and 1 GPU.

- *Experiment B* performs a detailed comparative evaluation of the EOT and the baseline technique (DAT) using different experimental setups.
- *Experiment C* focuses on evaluating the time to perform the offloading using various offloading techniques.

The experiments simulate 4 or 8 CPU cores and 1 or 2 GPUs. The implementation of the experiments can be extended to more than 8 CPU cores and more than 2 GPUs. However, such extensions are left for future work.

*1) **Experiment A**:* Fig. 4 illustrates the performance of the five offloading techniques in terms of the percentage of schedulable task sets with respect to the total number of tasks in the system (with utilization equals to 1). The NOT, MOT, SCT and SST show similar trend of schedulable task sets compared to the DAT and EOT, although there are some big differences between these two sets of techniques. This confirms that these four techniques (NOT, MOT, SCT and SST) focus on only part of properties for mitigating accelerator-intensive loads. Among these techniques, the SCT shows better results. The reason is that the SCT allocates a parallel segment with highest conversion ratio μ_i to CPU and the lowest conversion ratio to GPU. So, the SCT reorders the priority of the tasks and allocates them to the appropriate processing units. However, this can considerably change the order of the execution.

The DAT and EOT show the best results among the lot. More specifically, the EOT shows slightly better results than the DAT. It can be concluded that the use of accelerators is one of the best choices with respect to the schedulability of the task sets. However, the results of the EOT indicate that the improvement can be achieved by better utilizing all computing resources in the heterogeneous processors. Since the DAT and EOT are the best performing techniques, we focus only on them while performing a detailed comparative evaluation in the next simulation experiment.

*2) **Experiment B**:* In this simulation experiment, we consider how the EOT handles the schedulability of the task sets. Fig. 5 illustrates the percentage of schedulable task sets with

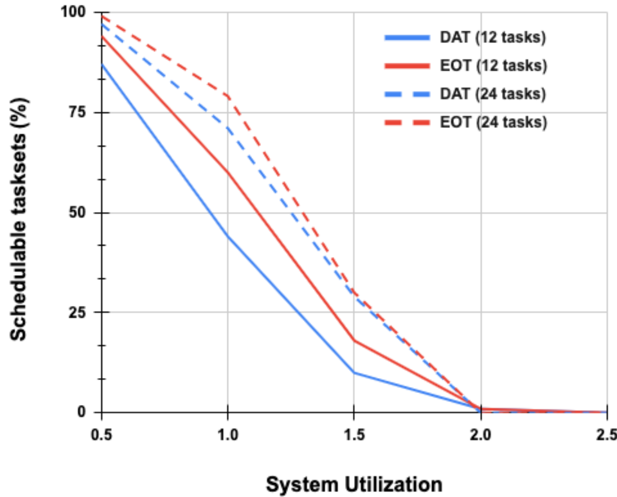

Fig. 5: Percentage of schedulable task sets with respect to the system utilization under the EOT and DAT techniques.

respect to the system utilization under the EOT and DAT techniques. The results indicate that the EOT performs better than the DAT, although the relative improvement is small. In the case of system utilization $U = 1$, the EOT with both 12 and 24 tasks shows the maximum improvements of 16% and 8% under dynamic scheduling.

Fig. 6 describes how the conversion ratio of parallel segment to sequential segment (i.e., μ_i) manipulates the schedulability of task sets. The horizontal axis expresses the maximum value that μ can take. We see that the EOT performs better than the DAT until the value of μ_i reaches 10. This means, our proposed framework can perfectly handle the accelerator-intensive workloads even if the accelerators compute 9 times faster than the host device.

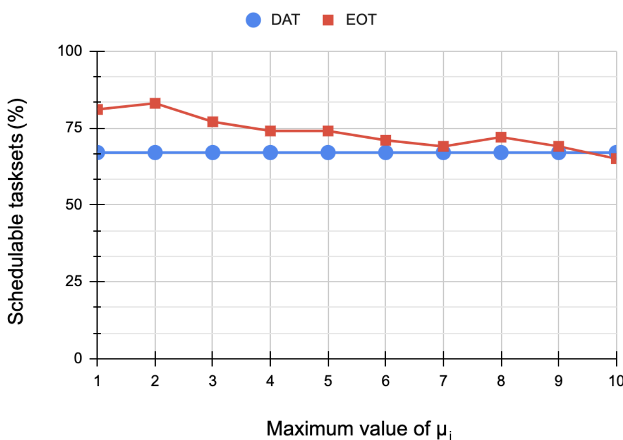

Fig. 6: Percentage of schedulable task sets with respect to variations in maximum value of μ_i under the EOT and DAT techniques.

Fig. 7 depicts how the variation in the ratio of the length of parallel and sequential segments, C_i^{GPU}/C_i^{CPU}, influences the schedulability of the task sets under the EOT and DAT techniques. Both techniques show a similar trend in the results; however, the EOT performs slightly better than the DAT. This confirms again that the use of accelerators for parallel segments is the best choice generally. There exists a slightly small improvement window between the essential use of accelerators and the total use of heterogeneous processors. When the length of parallel segments is equal to 75% of the entire execution time of the tasks, we observed up to 16% improvement in the schedulability of the task sets with respect to the DAT.

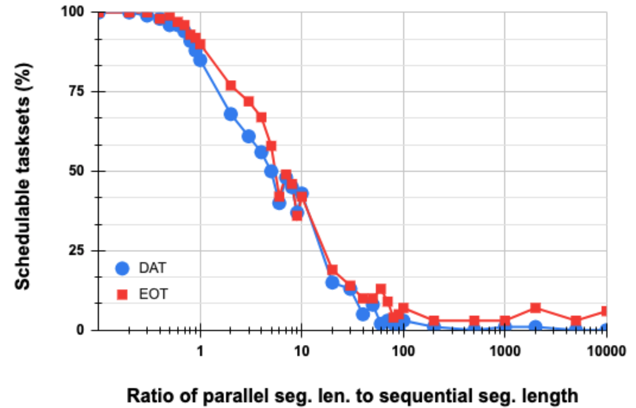

Fig. 7: Percentage of schedulable task sets with respect to the ratio of the length of parallel and sequential segments μ_i under the EOT and DAT techniques.

*3) **Experiment C**:* In this experiment, we extract the execution traces of the simulation of schedulable task sets. Table II shows the mean time to perform the offloading. The DAT shows the best mean time to perform offloading (22.33s), while the SCT gives the worst results (37.19s). The MOT and NOT more or less take the same time to perform the offloading. The reason is that these techniques select a parallel segment to allocate to the CPU, while the SCT needs to create a list, ordered by the μ_i, before selecting to allocate a parallel segment to the CPU. The SST shows the best results among the NOT, MOT, SCT and SST. In the SST, only one task can use the servers, which explains why the SST shows the best results among the four techniques.

Although the DAT performs the best in terms of the time to perform the offloading (22.33s), the EOT also performs nearly best (23.52s). This is because the EOT leverages the advantages of the NOT, MOT, SCT, and SST techniques.

VII. Conclusion

This paper presented a novel and efficient framework to allocate parallel segments of the accelerator-intensive workloads to non-accelerated processing units in the CPU-GPU heterogeneous processors under dynamic scheduling. Within the context of this framework, the paper proposed five offloading heuristic techniques for mitigating the accelerator-intensive workloads. The use of accelerators for parallel segments of the

TABLE II: Mean time to perform the offloading under various offloading techniques.

No.	Offloading Techniques	Experiment mean time
1	DAT	22.33s
2	NOT	34.49s
3	MOT	33.15s
4	SCT	37.19s
5	SST	28.75s
6	EOT	23.52s

tasks is crucial and is an excellent choice in most cases. In this regard, the paper showed that a considerable improvement can be achieved if all processing units in the heterogeneous processors are better utilized, i.e., by efficiently offloading some of the parallel segments to non-accelerated compute units. The evaluation results indicate that one of the proposed techniques, namely the efficient offloading technique, can achieve up to 16% improvement in schedulability of the task sets under dynamic scheduling compared to the non-offloading technique. It is worth to note that we have not optimized the SDS in this paper. In other words, the optimization of the SDS can improve the proposed framework and we consider it as future work. Another area of future work will focus on improvement of the proposed framework and offloading techniques using the pipelining technique. Furthermore, to expand our investigation to the use of multiple GPUs and/or different capacity of processing units entails another line of future work.

Acknowledgements: The work presented in this paper was supported by the the Swedish Knowledge Foundation via the DPAC and HERO projects, and the Swedish Governmental Agency for Innovation Systems (VINNOVA) via the DESTINE, PROVIDENT and INTERCONNECT projects.

REFERENCES

[1] J. Huang, "NVIDIA CEO Keynote," *GPU Technology Conference*, Oct. 2017.

[2] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, pp. 1038–1051, 2018.

[3] C. Margiolas and M. F. O'Boyle, "Portable and transparent software managed scheduling on accelerators for fair resource sharing," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, 2016, pp. 82–93.

[4] N. Tsog, M. Becker, F. Bruhn, M. Behnam, and M. Sjödin, "Static Allocation of Parallel Tasks to Improve Schedulability in CPU-GPU Heterogeneous Real-Time Systems," in *45th Annual Conference of the IEEE Industrial Electronics Society*, 2019, pp. 4516–4522.

[5] Y. Wen, Z. Wang, and M. F. O'boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms," in *2014 21st International conference on high performance computing (HiPC)*. IEEE, 2014, pp. 1–10.

[6] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE, 1998, pp. 4–13.

[7] M. Spuri and G. C. Buttazzo, "Efficient Aperiodic Service Under Earliest Deadline Scheduling," in *RTSS*, 1994, pp. 2–11.

[8] H. Zhu, S. Goddard, and M. B. Dwyer, "Response time analysis of hierarchical scheduling: The synchronized deferrable servers approach," in *32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 239–248.

[9] R. F. Freund and D. S. Conwell, "Superconcurrency: A form of distributed heterogeneous supercomputing," Naval Ocean Systems Center San Diego CA, Tech. Rep., 1991.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. IEEE, 2009, pp. 44–54.

[11] P. Czarnul and P. Rościszewski, "Optimization of execution time under power consumption constraints in a heterogeneous parallel system with GPUs and CPUs," in *International Conference on Distributed Computing and Networking*. Springer, 2014, pp. 66–80.

[12] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, 2011, pp. 17–30.

[13] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 401–412.

[14] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang, "An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads," in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 353–364.

[15] C. Basaran and K.-D. Kang, "Supporting preemptive task executions and memory copies in GPGPUs," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 287–296.

[16] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *2011 IEEE 32nd Real-Time Systems Symposium*. IEEE, 2011, pp. 57–66.

[17] E. Rossi, M. Damschen, L. Bauer, G. Buttazzo, and J. Henkel, "Preemption of the partial reconfiguration process to enable real-time computing with FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 2, pp. 1–24, 2018.

[18] G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: A framework for real-time GPU management," in *2013 IEEE 34th Real-Time Systems Symposium*. IEEE, 2013, pp. 33–44.

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

[20] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable FPGAs," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 1–12.

[21] S. Kato, K. Lakshmanan, Y. Ishikawa, and R. Rajkumar, "Resource sharing in GPU-accelerated windowing systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 191–200.

[22] Y.-S. Chen, H. C. Liao, and T.-H. Tsai, "Online real-time task scheduling in heterogeneous multicore system-on-a-chip," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 1, pp. 118–130, 2012.

[23] S. Baruah, "Resource-Efficient Execution of Conditional Parallel Real-Time Tasks," in *European Conference on Parallel Processing*. Springer, 2018, pp. 218–231.

[24] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, 2014, pp. 3–12.

[25] F. C. Bruhn, N. Tsog, F. Kunkel, O. Flordal, and I. Troxel, "Enabling Radiation Tolerant Heterogeneous GPU-based Onboard Data Processing in Space," *CEAS Space Journal*, vol. 12, no. 4, pp. 551–564, 2020.

[26] N. Tsog and M. Larsson, "Time Predictability of GPU Kernel on an HSA Compliant Platform," 2016.

[27] S. K. Baruah, "Task Partitioning Upon Heterogeneous Multiprocessor Platforms," in *IEEE real-time and embedded technology and applications symposium*. Citeseer, 2004, pp. 536–543.

[28] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen *et al.*, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed computing*, vol. 61, no. 6, pp. 810–837, 2001.

[29] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1-2, pp. 129–154, 2005.

[30] N. Tsog, M. Behnam, M. Sjödin, and F. Bruhn, "Intelligent data processing using in-orbit advanced algorithms on heterogeneous system architecture," in *the IEEE Aerospace Conference*, 2018, pp. 1–8.

[31] N. Tsog, M. Sjödin, and F. Bruhn, "Advancing on-board big data processing using heterogeneous system architecture," in *ESA/CNES 4S Symposium 4S 2018, 28 May 2018, Sorrento, Italy*, 2018.