Static Allocation of Parallel Tasks to Improve Schedulability in CPU-GPU Heterogeneous Real-Time Systems

Nandinbaatar Tsog[1], Matthias Becker[2], Fredrik Bruhn[1,3], Moris Behnam[1], Mikael Sjödin[1]
[1]*Mälardalen University*, Sweden, firstname.lastname@mdh.se
[2]*KTH Royal Institute of Technology*, Sweden, mabecker@kth.se
[3]*Unibap AB (publ.)*, Sweden, f@unibap.com

***Abstract*—Autonomous driving is one of the main challenges of modern cars. Computer visions and intelligent on-board decision making are crucial in autonomous driving and require heterogeneous processors with high computing capability under low power consumption constraints. The progress of parallel computing using heterogeneous processing units is further supported by software frameworks like OpenCL, OpenMP, CUDA, and C++AMP. These frameworks allow the allocation of parallel computation on different compute resources. This, however, creates a difficulty in allocating the right computation segments to the right processing units in such a way that the complete system meets all its timing requirements. In this paper, we consider pre-runtime static allocations of parallel tasks to perform their execution either sequentially on CPU or in parallel using a GPU. This allows for improving any unbalanced use of GPU accelerators in a heterogeneous environment. By performing several heuristic algorithms, we show that the overuse of accelerators results in a bottle-neck of the entire system execution. The experimental results show that our allocation schemes that target a balanced use of GPU improves the system schedulability up to 90%.**

I. Introduction

Modern cars face several cases to be solved, such as environmentally friendly vehicles, connected vehicles, and self-driving cars. For example, while identifying obstacles using computer vision applications or making more smart decisions by on-board AI (artificial intelligence) applications, modern cars should be energy efficient. In order to cope with these challenges, thus, efficient energy consumption and intelligent on-board processing are crucial. With the increasing demand on processing capability with low power-consumption, the trend of processing units in real-time systems has started to shift from single core to multi- and many-core CPUs as well as heterogeneous processing units [1]. For example, a CPU is preferred for sequential computation while a GPU shows its advantages in parallel numerical computation.

Academic work on sequential and parallel computation is broadly conducted in the real-time and high performance computing communities. In heterogeneous computing [2], [3], a task is often considered as a sequence of multiple segments. A task segment can represent either sequential execution or parallel execution, where the same function is applied on different parts of the data as shown in Figure 1. A parallel segment can be executed by different processing units (not only GPU), however, in this paper we only consider GPU and CPU resources. Furthermore, the allocation of parallel segments of the program to compute resources is typically done at design time. However, allocating all parallel segments to the same resource might over-restrict the systems. In high performance computing and supercomputer community, distributed computing is considered in order to decrease the overuse of the same resource [3].

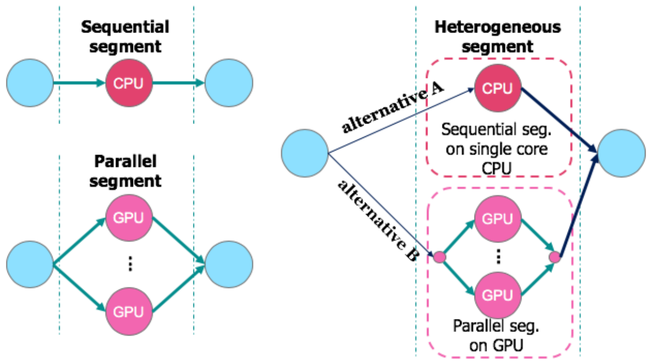

Fig. 1. The execution alternatives of a parallel segment of tasks

In this work, we consider an extension of the fork-join task model [4], by adopting the alternative executions of parallel segments since this extended model has been introduced in the real-time community just recently [5]. A parallel segment can either be mapped to a GPU for parallel execution (alternative B) or to a CPU for sequential execution of the same code segment (alternative A), see Figure 1. This then allows to take the characteristics of the execution on the different processing units into account during the system design phases where mapping decisions are taken.

Traditionally, parallel computations are often allocated to the GPU resources as it provides better performance compared to sequential execution on the CPU. However, the overuse of GPUs may end up in a bottle-neck situation [6]. Therefore, a study of how we can balance the allocation of parallel tasks to different processing units is warranted and important.

A. Contributions

In this work, we study how static allocation of parallel tasks can improve schedulability of task sets in GPU accelerated

Swedish special characters (å, ä, ö) in the email addresses are replaced by a and o.

real-time systems. This allows to eliminate the bottle-neck caused by overuse of GPUs. Our main contributions in this paper are:

- We show that the overusing of GPU might bring a negative impact to the schedulability of real-time systems.
- We tackle this problem by offloading GPU computation to CPU. In other words, we conduct pre-runtime static allocations of the parallel segments either on CPU or GPU.
- We adapted the following heuristic approaches using synthetic workloads: Non-Greedy Resource Allocation Heuristic Approach (NHA), Speedup Classifier based Heuristic Approach (SHA), and Min-Min Approach (MMA). We show that the algorithms based on these approaches improve the schedulability of task sets compared to their default task mapping on the GPUs using Baseline Task Set (BTS) approach, see Section V-B).
- Our synthetic experiments show up to 90% of improvement of the schedulability of task sets depending on the platform size compared to the default task mapping on the GPUs (BTS).

B. Organization

In the rest of this paper, we motivate this paper in Section II. Section III presents a description of our system model, followed by detailed explanation of the task model. Heuristic task allocation approaches are introduced in Section IV. In Section V, we describe experiments and their setups and report our experimental evaluation. We discuss the related work in Section VI and lastly, conclusions are presented in Section VII.

II. Motivation

Due to the following two reasons, we strongly consider the importance of balanced use of CPU and GPU in real-time systems with the partitioned fixed priority preemptive scheduling for CPU and the fixed priority non-preemptive scheduling for GPU.

- As the GPU is a single shared resource under the fixed priority non-preemptive scheduler, a higher priority task could be blocked by lower priority tasks. The blocking is well known in real-time community and it appears in many ways such as priority inversion problem.
- Interference from higher priority tasks extends the response time of a lower priority task that may result in the deadline miss.

As shown in Figure 2, assume a scenario for the first case above by considering the scheduling of three tasks, τ_h, τ_m and τ_l, with high, middle and low priorities, respectively. The system has 2 CPU cores and a single GPU, and follows rate-monotonic priority assignment on CPU and non-preemptive on GPU. We assume that task τ_h has 3 segments with sequential, parallel, sequential order. The worst case execution times (WCETs) of each segment are 2, 1, and 3 time units. As we focus on task τ_h, we do not consider all the segments of tasks τ_m and τ_l. Only necessary segments of these tasks are given as follows. The first segment of task τ_m is sequential with 3 time units of WCET. The first two segments of τ_l are sequential (WCET: 1 time unit) and parallel (WCET: 11 time units).

Then, task τ_l releases at 0 for 1 time unit on CPU core 2 and executes then with 11 time units on GPU. Task τ_m starts at 1 for 3 time units on CPU core 1 and tries to run on GPU. However, this execution will be blocked by task τ_l. Later, task τ_h releases at 4 on CPU core 1 for 2 time units. At 6, task τ_h tries to run on GPU, however, GPU is still occupied with task τ_l. Task τ_h starts running on GPU at 12 for 1 time unit and runs back on CPU core 1 from 13 for 3 time units. Therefore, task τ_h completes at 16 and the response time is 12.

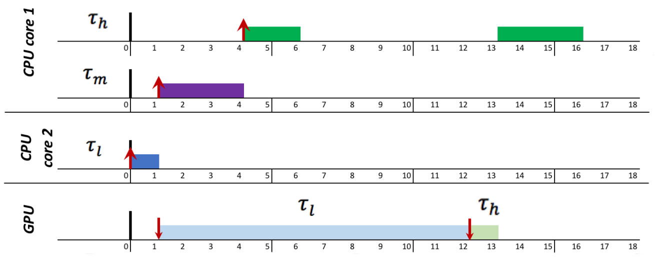

Fig. 2. The execution of parallel segment on GPU

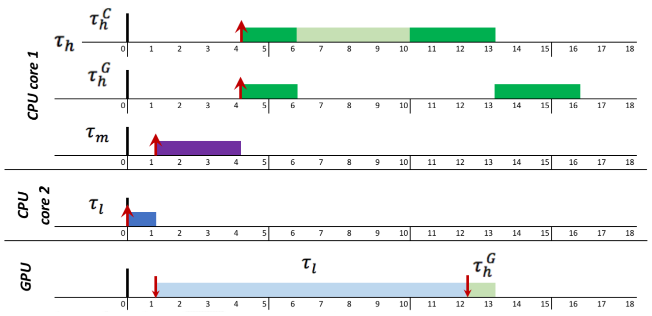

Fig. 3. The execution of parallel segment either on GPU or CPU

Now we consider the alternative executions of parallel segments shown in Figure 3. In addition to the previous case, we assume that the WCET of the parallel segment of task τ_h on CPU is 4 time units, which is 4 times longer than the WCET on GPU. In order to identify both alternatives, we use the notation of τ_h^C and τ_h^G, which are the executions of parallel segment on CPU and GPU, respectively. In Figure 3, we can see that the previous case is described as τ_h^G. Then, let us consider the case of τ_h^C. Until 6, the executions of both τ_h^G and τ_h^C are the same. As we know GPU is busy with τ_l at 6, τ_h^C can be executed on CPU. This execution finishes at 10, and the third segment of task τ_h is executed on CPU continuously. In this case, task τ_h finishes at 13 and its response time is 9. This means that the execution on CPU (τ_h^C) has shorter response time than the execution on GPU (τ_h^G).

III. System and Task Model

A. System model

We consider a system S, that consists of a task set Γ with n tasks. A hardware platform has m identical CPUs $\{P_m^{CPU}\}$

and a single GPU device P^{GPU}.

$$S =< \Gamma, \{P_m^{CPU}\}, P^{GPU} >$$

For CPU scheduler, the partitioned fixed priority preemptive scheduling technique is assumed. GPU allows to execute tasks with non-preemptive fixed priority scheduling.

B. Task Model

A task $\tau_i \,\epsilon\, \Gamma$ is represented by the fork-join task model, where parallel workload is modelled by fork/join segments. τ_i can be characterized by the tuple $\{S_i, D_i, T_i\}$, where D_i is the relative deadline of the task and T_i is the period of the task. S_i is composed of a finite sequence of l different execution segments $\{S_{i,1}, S_{i,2}, \dots, S_{i,l}\}$, where $l \in \mathbb{N}$ (see Figure 4). Each of the segments $S_{i,j} \in S_i$ represents a number of k sub-tasks $\tau_{i,j}^1, \dots, \tau_{i,j}^k$, where $k \in \mathbb{N}$, and has an assigned Worst-Case Execution Time (WCET) for each segment, denoted by $C_{i,j}$. We define a sequential segment for $S_{i,j}$ where k=1 (having a single sub-task), and a parallel segment for $S_{i,j}$ where k >1 (having multiple sub-tasks). We further require that the first and last segment of the task have a segment with only one sub-task. During execution, the sub-tasks of a segment $S_{i,j}$ can only be released once all sub-tasks of prior segments $S_{i,j-1}$, completed their execution. As this model defines the application, at this stage, no allocation to the different compute resources (CPU or GPU) is included and the WCET estimates are generally not known. We assume the allocation of the sequential segments on CPU is predefined. We are aware that changing this allocation might have a big impact on the performance and combining this with our solutions might provide better result. However, we leave this part as future work.

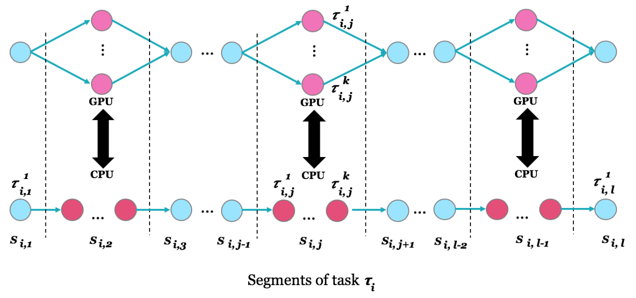

Fig. 4. Execution segments of task τ_i and their allocations on the different resources

As the main focus of this work is the allocation of computation (parallel segments) to the different hardware types, the model is further extended. A parameter $h_{i,j}$ is added for each segment $S_{i,j} \in S_i$ of a task τ_i. $h_{i,j}$ encodes the allocation decision, i.e. $h_{i,j} = \texttt{CPU}$ or $h_{i,j} = \texttt{GPU}$, where CPU and GPU are different constants.

The execution time $C_{i,j}$ is then dependent on the selected compute resource, and is represented as:

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

This represents the parallel execution of a segment, when assigned to the GPU (denoted by $C_{i,j}^{\texttt{GPU}}$) and the sequential execution of all k sub-tasks on the CPU(denoted by $C_{i,j}^{\texttt{CPU}}$). It also needs to be noted that typically execution on the GPU requires miscellaneous operations (such as copying of data). To model this, the execution time on the GPU can further be divided into $C_{i,j}^{\texttt{GPU}} = G_{i,j}^m + G_{i,j}^C$. Where $G_{i,j}^m$ represents miscellaneous computation and $G_{i,j}^C$ represents actual computation on the GPU.

To ease the later presentation of our approach we define $C_i^{\texttt{CPU}}$ (or simply C_i) and $C_i^{\texttt{GPU}}$ (or simply G_i) as the total execution time that the task τ_i requires from CPU and GPU respectively:

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

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

To give a estimate if a task τ_i is CPU-heavy or GPU-heavy, a metric is used as conversion ratio between the two options. This conversion ratio is denoted by μ_i and gives the ratio of required execution on CPU to GPU for all *parallel segments* (i.e. segments that have more than 1 sub-task, $k > 1$):

$$\mu_i = \frac{\sum\limits_{1 \le j \le l,\ k>1,\ h_{i,j} = \texttt{CPU}} C_{i,j}}{\sum\limits_{1 \le j \le l,\ k>1,\ h_{i,j} = \texttt{GPU}} C_{i,j}} \tag{4}$$

Smaller values of μ_i indicate that the benefit of execution on the GPU is smaller compared to larger values of μ_i.

As all segments $\tau_{i,j}$, where $k = 1$, represent purely sequential segments of execution, they must be executed on the CPU and $h_{i,j} = \texttt{CPU}$. For all other segments $h_{i,j}$ depends on the concrete allocation.

IV. Heuristic Task Allocation Approaches

The response time of a task can be improved by selectively assigning parallel segments to CPU or GPU as highlighted in Section II. The allocation of parallel segments is considered as one dimensional bin packing problem since it is defined as a problem of finding an optimal allocation of the parallel segments to the suitable processing resources. However, finding an optimal solution ends up with time complexity, which may be described with a year-unit. Thus, improved exhaustive algorithms and heuristic algorithms should be taken into account in order to decrease the search space and to find some reasonable results, respectively. Due to the limited space, in this section, we only skim an overview of heuristic approaches to allocate the tasks into their preferable processors instead of concerning the optimal allocation. The aim of these approaches is to improve the schedulability of a given task set. These approaches take a task set as an input and returns the schedulability of the task set. We assume that all the parallel segments of tasks are allocated to GPU by default.

A. Non-Greedy Resource Allocation Heuristic Approach (NHA)

NHA is a simple allocation approach, which is intended to reveal the importance of the balanced use of CPU and GPU. This approach performs the following steps.

- Step 1. Check the schedulability of the input task set by using a schedulability analysis test. This approach returns *schedulable* and stops if the task set is *schedulable*.
- Step 2. Identify a task that misses its deadline.
- Step 3. Return *not schedulable* if the task has been allocated on CPU already. Otherwise, allocate the task to CPU and goto Step 1 again.

B. Speedup Classifier based Heuristic Approach (SHA)

This approach is inspired by [3]. As illustrated in Figure 5, the idea of SHA is based on a queue of GPU-using tasks which is sorted by a speedup classifier. SHA follows the steps below.

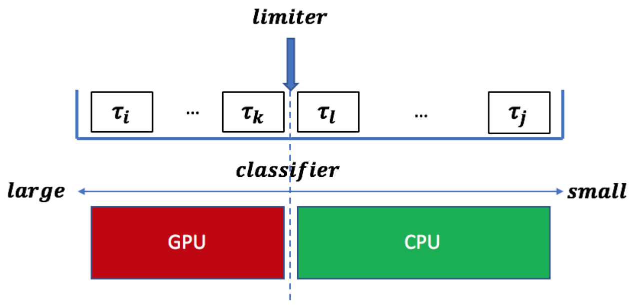

Fig. 5. A sorted task queue based on speedup classifier

- Step 1. Add tasks to a queue.
- Step 2. Sort them in order of speedup classifier as the task with the largest classifier is first and the task with the smallest classifier is last.
- Step 3. Allocate the parallel segments of tasks on the left side of limiter on GPU and the parallel segments of tasks on the right side of limiter on CPU. In order to find the suitable limiter, we perform the following different cases that 0%, 20%, 40%, 60%, 80% and 100% of the tasks are allocated on the left side of the limiter.
- Step 4. To check and returns the schedulability test of the task set.

In [3], the speedup classifiers are calculated by using a support vector machine (SVM). However, using machine learning algorithms may require longer allocation time, which will be a disadvantage in real-time systems. Therefore, instead of the speedup classifiers based on SVM, we consider the conversion ratio of parallel segment(μ_i), priority, and utilization of tasks. We call the algorithms used these classifiers as SHA-μ, SHA-prio, and SHA-util, respectively.

C. Min-Min Approach (MMA)

Braun et al. [7] report the Min-min approach shows the second best results among eleven static heuristics for mapping tasks onto heterogeneous distributed computing systems. The best heuristic is GA (Genetic Algorithms). However, similar to the SVM based speedup classifiers, we have adapted GA as its allocation time can be longer. In this paper, we consider the Min-min fashioned approach which is described in the steps below.

- Step 1. Initially, there is no task allocated to any processors.
- Step 2. All the CPU-using tasks are allocated to each processor which is allocated to when tasks are generated. All the GPU-using tasks are placed in a queue in order of priority.
- Step 3. Pick the highest priority task from the queue and calculates the two response times of this task using CPU or GPU for the parallel segment, respectively. If there is no more task in the queue and all the response-times are no greater than the deadlines, we say the task set is schedulable and the solution is provided.
- Step 4. Allocate the parallel segments of the task either on CPU or GPU according to the lowest response time of these cases.
- Step 5. Stop the algorithm if the systems is not schedulable when the task is allocated to the suitable processor(s) in Step 4. Otherwise, remove the task from the queue and go to Step 3.

V. Synthetic Experiments

In order to evaluate a wide range of application parameters with our proposed approach, synthetic experiments are performed.

TABLE I
Initial configuration of task set generation

Parameters	Values
Number of CPU cores (N_p)	4, 8
Number of tasks (n)	$[2N_p, 6N_p]$
Task utilization (U_i)	[0.1, 0.2]
Task period and deadline ($T_i = D_i$)	[30, 500]ms
Percentage of GPU-using tasks	[10, 30]%
Ratio of GPU segment len. to normal WCET (G_i/C_i)	[10, 30]%
Number of parallel segments per task (η_i)	0 or 1-3
Ratio of misc. operations in $G_{i,j}$ ($G^m_{i,j}/C^{GPU}_{i,j}$)	[10, 20]%
GPU server overhead (ϵ)	$50\mu s$
Conversion ratio of parallel segments (μ_i)	[3-10]

A. Task set generation

The mechanism of task generation is based on the UUniFast algorithm, which is proposed by Bini and Buttazzo [8]. Necessary configuration values are described in Table I. First, in order to generate both alternatives (the execution either on CPU or GPU), we generate the WCET of parallel segments, which is generated by using the ratio of GPU segment length to sequential segment's WCET. Then, we create the WCET of the sequential execution of the parallel segment by using a conversion ratio of parallel segments, μ_i, which is in default a random number between 3 and 10 brought from the experimental studies from [9], [10].

B. Comparative algorithms

In order to evaluate our idea of the removal of bottlenecks of CPU-GPU, we consider 6 comparative algorithms (*BTS (Baseline Task Set)*, *NHA*, *SHA-μ*, *SHA-prio*, *SHA-util*, and *MMA*) using the proposed heuristic approaches in Section IV. *Baseline Task Set (BTS)* is a task set that all the parallel segments of the tasks are allocated on GPU. We choose BTS as a baseline for comparison investigation. The other algorithms are based on the proposed approaches.

The input to the approaches is a set of tasks that must fulfill the input requirements of the task model (Section III-B) such as segments, deadlines and periods. As we defined in III-B, the sequential segment of tasks is preemptible and self-suspending [6], [11]. Further, a parallel segment of tasks is stored in the priority-based GPU scheduler queue. The current executing parallel segment on the GPU is non-preemptive. In other words, a parallel segment can only be delayed by the higher priority parallel segments when this segment is on the GPU scheduler queue. Any schedulability analyzing test [11] for the given execution model can be used together with the proposed heuristic algorithms. We assume that the schedulability analysis returns the highest priority task which misses its deadline among tasks in the task set. Otherwise, the algorithm finishes its schedulability analysis and classifies the task set as schedulable.

C. Experiment setup

In the experiments, we used 10,000 randomly generated task sets in general. The experiments are based on the parameters as they shown in Table I except for the parameters that are varied in the respective experiment. To decide upon the schedulability of the task sets under the mapping that is created by our proposed approaches, the schedulability analysis for server-based approaches of [11], [12] are used.

D. Result

In this section, we describe the following 3 groups of experiments. *Experiment A* focuses on the understanding of the balanced use of CPU-GPU and *Experiment B* is a comparison study between 5 comparative algorithms under different experiment setups. *Experiment C* is focusing on the experiment time for the heuristic algorithms. The experiments are targeting a system that has either 4 or 8 CPU cores and 1 GPU. Due to the limited space, we show only the results with 24 tasks on 4 CPU cores.

Experiment A In this experiment, we focus on the change of schedulable task sets as the task set changes from 0% of GPU-using tasks (i.e., 100% of CPU-using tasks) to 100% of GPU-using tasks (i.e., 0% of CPU-using tasks). The peak value of the change describes the balanced use of CPU-GPU. For example, in Figure 6, the curve for $\mu_i = 5$ gets the peak value of around 70% of schedulable tasksets at 20% of GPU-using tasks and 80% of CPU-using tasks. Furthermore, the schedulability of task sets with the different $\mu_i = 5$ and $\mu_i = 15$ results in about 0% and 97%, respectively. In other words, in case of $\mu_i = 5$, the execution time on GPU is 3 times

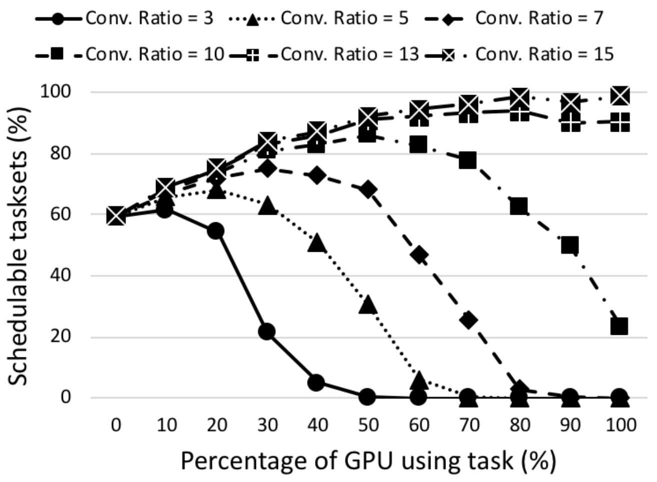

Fig. 6. Schedulable task sets w.r.t. the fixed conversion ratio (4 CPU cores, 24 tasks)

smaller than the case of $\mu_i = 15$ where the total execution times are the same for both cases. Hence, this result could be explained that the tasks with the shorter execution time on GPU can block and interfere the other tasks less compared to the tasks with the longer execution time on GPU. This shows the importance of the balanced use of CPU-GPU. We note that this experiment uses NHA algorithm in order to understand and optimize task sets.

Experiment B Figure 7 illustrates the percentage of schedulable task sets with respect to the percentage of GPU using task with the default settings. We see that MMA is the best performing heuristics, since the percentage of schedulable task set is 100% in all the cases. NHA is the second best heuristics, and SHA-prio and SHA-util follow. In this case, SHA-μ did not perform well compared to the other heuristics. Here, in 100% of GPU using task, we confirm that NHA and MMA perform about 70% and 90% better than BTS, respectively.

Figure 8 shows the results when the range of μ_i has been extended from [3-10] to [3-100]. This change, obviously, worsens the execution of parallel segment on the CPU as it may take 100 times longer on CPU than GPU. In this case, we see only NHA improves BTS. Further, the algorithms SHA-prio, SHA-util and MMA could not succeed for the allocation. Because, these algorithms allocate tasks with longer execution times (about 100 times) on CPU compared to the tasks in BTS. Moreover, all these algorithms would not consider the results of BTS whether they are schedulable or not. Hence, the results of these algorithms are lower than BTS (see Figure 8). The schedulability of task set could be improved when these algorithms include the knowledge of whether BTS is schedulable or not although the experiment time gets longer. For example, the schedulability result of BTS is the same in both Figures 7 and 8. However, these algorithms result worse than BTS in Figure 8, and better in Figure 7 On the other hand, NHA improves the results all the time compared to BTS as NHA is based on the results of BTS.

Experiment C Table II shows the experiment time of the heuristic algorithms for a task set measured 10,000 times. The

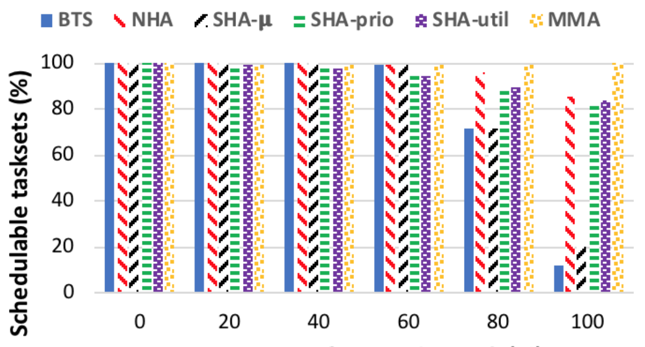

Fig. 7. Schedulable task sets w.r.t GPU using task with the default settings

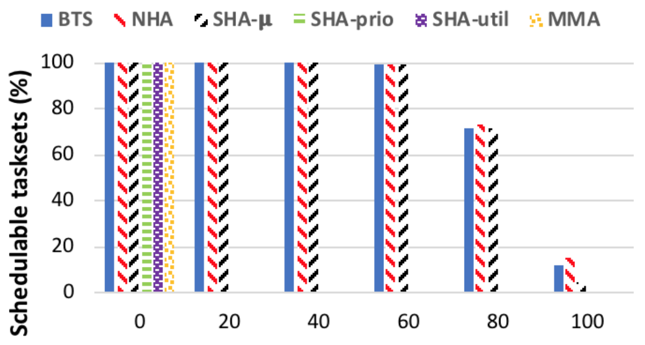

Fig. 8. The extended range of μ_i (between 3-100)

experiment time consists of the times of the task generation, task allocation and schedulability analysis for a task. Max, median, mean and min values of 99.9th percentile of the experiment time are shown in Table II.

We can see that the median value of NHA (40us) is the best among all the heuristic algorithms. SHA-μ (55us), SHA-prio (51us), and SHA-util (51us) are in the same level although SHA-μ takes a bit longer time compared to other 2 algorithms. MMA requires the longest experiment time among all the algorithms while MMA gives the best schedulability results in most of the scenarios. In conclusion of the experiment C, we could say that NHA is the best choice from the time-wise.

VI. Related work

Heterogeneous computing is well studied in high performance community, especially, in supercomputers as an extension of the distributed computing [3], [13]–[15]. However, these techniques are not investigated targeting the real-time embedded systems. Furthermore, the techniques are mostly focused on the distributed heterogeneous systems, not on the heterogeneous systems included in the system on chips although their architecture may be similar to each other.

There exist several approaches considering the use of GPU in real-time embedded systems. Kato et al. introduced TimeGraph [16], RGEM [17] and Gdev [18] along with zero-copy I/O processing for low-latency GPU computing [19]. In addition, self-suspending task techniques [6], [20] are broadly studied in real-time systems, and they are applicable to GPU accelerated real-time systems. Most of these works

TABLE II
The experiment time for heuristics algorithms for a task set

No.	Heuristic Algorithm	*Experiment time [us]*			
		max	median	mean	min
1	BST	71.0	25.5	27.0	19.0
2	NHA	121.0	40.0	43.9	22.0
3	SHA-μ	134.0	55.0	58.2	34.0
4	SHA-prio	116.0	51.0	53.6	42.0
5	SHA-util	103.0	51.0	53.0	43.0
6	MMA	930.0	419.0	433.8	249.0

consider compensating the limitation of early existing GPU hardware and device drivers such as a zero-copy technique for accelerators' memory and splitting tasks into smaller chunks for allowing preemption. However, these limitations will be solved by coming new technologies such as unified memory, zero-copy and preemption technologies in CUDA [21] and Heterogeneous System Architecture (HSA) [9], [10], [22]. Furthermore, the works of Elliott et al. [23], [24] and Kim et al. [11], [12] consider worst-case timing behavior in GPU accelerated real-time systems. These works could be used as the timing analysis tool.

The bin-packing problem is one of the most important optimization problems. In our work, the problem is to find an optimal solution for the allocation of tasks to different processing units. The study of the bin-packing problem has been done widely in parallel processing [25]–[28] and real-time systems [29]. However, due to its NP-hard nature, it is common to introduce heuristic approaches to figure out the system in an affordable time. Moreover, there are many works regarding the resource mapping in heterogeneous platforms such as [7], [13], [30], [31]. In this paper, we adopt the heuristic approaches from [3], [7] in order to understand how the behavior of the alternative execution (either on CPU or GPU) of parallel segment affects the schedulability of task sets.

VII. Conclusions

In this paper, we target the mapping of parallel segments to either the GPU or CPU. While GPU resources can boost the performance, heavy usage can result in a bottleneck for the system. This affects on one hand tasks that want to access the shared GPU resource, as they can be blocked by other GPU segments, and on the other hand tasks that receive additional interference due to blocking by the GPU-handler task that is executed on the CPU. Thus, always executing parallel code segments on the GPU potentially degrades the system performance. We demonstrate that selective mapping of parallel segments to either CPU or GPU can improve the system performance. Heuristic algorithms are described to select the respective compute resource for such tasks. Synthetic evaluations reveal that our proposed heuristics are able to improve the schedulability of task sets up to 90% compared to task sets where no mapping decisions are taken.

Future work will focus on runtime assignment of parallel segments to CPU or GPU. Furthermore, we hope to expand our investigation to the use of multiple GPUs or any other accelerators.

References

[1] 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, 2019.

[2] 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.

[3] 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.

[4] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *22nd International Conference on Real-Time Networks and Systems (RTNS)*. ACM, 2014, pp. 3:3–3:12.

[5] S. Baruah, "Resource-efficient execution of conditional parallel real-time tasks," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Springer International Publishing, 2018, pp. 218–231.

[6] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, "Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions," *Leibniz Transactions on Embedded Systems*, vol. 5, no. 1, pp. 02–1–02:20, 2018.

[7] 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.

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

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

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

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

[12] ——, "A server-based approach for predictable gpu access with improved analysis," *Journal of Systems Architecture*, vol. 88, pp. 97–109, 2018.

[13] D. Grewe and M. F. OBoyle, "A static task partitioning approach for heterogeneous systems using opencl," in *International Conference on Compiler Construction*. Springer, 2011, pp. 286–305.

[14] 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.

[15] H. Zhou and C. Liu, "Task mapping in heterogeneous embedded systems for fast completion time," in *2014 International Conference on Embedded Software (EMSOFT)*. IEEE, 2014, pp. 1–10.

[16] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *USENIX Conference on USENIX Annual Technical Conference (USENIXATC)*. USENIX Association, 2011, pp. 2–2.

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

[18] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," in *USENIX Conference on Annual Technical Conference (USENIXATC)*. USENIX Association, 2012, pp. 37–37.

[19] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy i/o processing for low-latency gpu computing," in *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, April 2013, pp. 170–178.

[20] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, D. de Niz, and G. von der Brüggen, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," *Real-Time Systems*, Sep 2018.

[21] M. Harris, ""Unified Memory for CUDA Beginners." June 19, 2017." available: https://devblogs.nvidia.com/unified-memory-cuda-beginners/ [Oct 16, 2018].

[22] HSA Foundation, "Heterogeneous system architecture," available: http://www.hsafoundation.com/ [Oct 16, 2018].

[23] G. A. Elliott, B. C. Ward, and J. H. Anderson, "Gpusync: A framework for real-time gpu management," in *34th IEEE Real-Time Systems Symposium (RTSS)*, Dec 2013, pp. 33–44.

[24] G. A. Elliott and J. H. Anderson, "Globally scheduled real-time multiprocessor systems with gpus," *Real-Time Systems*, vol. 48, no. 1, pp. 34–74, Jan. 2012.

[25] J. O. Berkey, "Massively parallel computing applied to the one-dimensional bin packing problem," in *Proceedings., 2nd Symposium on the Frontiers of Massively Parallel Computation*, Oct 1988, pp. 317–319.

[26] E. G. Coffman, M. R. Garey, and D. S. Johnson, *Approximation Algorithms for Bin-Packing — An Updated Survey*. Springer Vienna, 1984, pp. 49–106.

[27] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[28] J. D. Ullman, "Np-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, no. 3, pp. 384–393, Jun. 1975.

[29] D. De Niz and R. Rajkumar, "Partitioning bin-packing algorithms for distributed real-time systems," *International Journal of Embedded Systems*, vol. 2, no. 3-4, pp. 196–208, 2006.

[30] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 692–707, 2010.

[31] A. H. Alhusaini, V. K. Prasanna, and C. S. Raghavendra, "A framework for mapping with resource co-allocation in heterogeneous computing systems," in *Heterogeneous Computing Workshop, 2000.(HCW 2000) Proceedings. 9th*. IEEE, 2000, pp. 273–286.