



Task-to-accelerator mapping for predictable OpenMP applications[☆]

Mohammad Samadi^a,^{*}, Tiago Carvalho^a, Luís Miguel Pinho^a, Sara Royuela^b

^a Polytechnic Institute of Porto & INESC TEC, Porto, Portugal

^b Barcelona Supercomputing Center, Barcelona, Spain

ARTICLE INFO

Dataset link: <https://github.com/m-samadi/task-to-accelerator-mapping/>

Keywords:

OpenMP

Accelerator

Mapping process

Heuristics

Multi-criteria decision analysis

ABSTRACT

OpenMP offers a high-level, portable programming model that simplifies the expression of parallelism and offloading in heterogeneous platforms, making it an ideal framework for managing task execution across CPUs and GPUs. However, as applications increasingly require predictability and multi-GPU execution, the lack of predictability in current OpenMP scheduling mechanisms poses significant challenges. Irregular workloads, variable data transfers, and dynamic resource contention often lead to inconsistent response times and reduced timing determinism. This unpredictability limits system efficiency and reliability, particularly in performance-critical and real-time domains. To overcome the challenge mentioned above, this paper proposes an efficient task-to-accelerator mapping (i.e., assigning tasks to GPU devices for execution) for multi-GPU OpenMP applications. The mechanism includes an offloading manager that operates a global queue and multiple local queues, and different heuristic algorithms that first allocate tasks to the most appropriate queues and then select the most suitable tasks from each queue. The proposed heuristics aim to balance queue loads and enhance the work-conserving property of the mapping process to minimize response time and response-time variability. The paper also presents a thorough evaluation using both synthetic graphs and graphs representative of real-world applications, showing that the proposed mapping method outperforms the other relevant mapping methods in terms of application response time and response-time variability.

1. Introduction

Domains such as the Internet of Things (IoT) and Cyber-Physical Systems (CPS) include applications with demanding performance and timing requirements, as they often must process large amounts of data while meeting stringent latency and determinism constraints to ensure safety [1]. To meet such demands, modern platforms increasingly rely on heterogeneous systems that combine multicore CPUs with multiple GPUs and other accelerators, providing the computing resources needed to efficiently handle complex workloads.

The coexistence of diverse processing units introduces new difficulties in ensuring *predictable execution* and *reliable timing analysis* compared to homogeneous systems [2]. Critical aspects include competition for shared resources, the black-box nature of caches, and the variable behavior of scheduling mechanisms. This unpredictability complicates the timing analysis of parallel applications, which is fundamental in safety-critical and real-time domains. Consequently, the task-to-accelerator mapping process plays a critical role by determining when and where ready tasks are executed, directly impacting both response time and timing variability.

Existing GPU scheduling techniques only partially address these issues. Many approaches are designed for single-GPU systems and fail to scale efficiently to multi-GPU environments. Others rely on complex runtime methods that may improve throughput but further reduce predictability. Blocking scheduling approaches compromise resource utilization and increase response time by preventing work-conserving behavior. Beyond scheduling itself, adapting applications to leverage multi-GPU capabilities without compromising programmability and portability remains a major open challenge [3–10].

Parallel programming models like OpenMP provide an appealing solution by abstracting low-level architectural details [11,12] through a standardized, portable interface for expressing task-level parallelism and heterogeneous offloading. While initially designed for High-Performance Computing (HPC), previous work has demonstrated the suitability of OpenMP for real-time applications [13,14]. The Directed Acyclic Graph (DAG) that captures the parallelism of OpenMP tasks enables systematic reasoning about task dependencies, parallel execution, and timing properties. The inclusion of specialized *target* tasks further enables holistic analysis of both CPU and accelerator tasks, thereby facilitating end-to-end reasoning in heterogeneous systems.

[☆] This article is part of a Special issue entitled: 'AEIC2026' published in Journal of Systems Architecture.

^{*} Corresponding author.

E-mail addresses: mmasa@isep.ipp.pt (M. Samadi), tdc@isep.ipp.pt (T. Carvalho), lmpp@isep.ipp.pt (L.M. Pinho), sara.royuela@bsc.es (S. Royuela).

Despite these advantages, significant challenges remain before OpenMP can be reliably used in time-constrained domains. Advanced GPU scheduling methods for OpenMP often focus on single-GPU systems [10] or require extensive user intervention to exploit multi-GPU systems efficiently [15]. Moreover, asynchronous target tasks are needed to avoid the *busy-wait* forced by synchronous execution on the GPU, as it reduces system efficiency by losing the work-conserving nature of the scheduling process [3]. Finally, most OpenMP implementations (e.g., LLVM) do not consider application and system-wide information to implement a mapping that minimizes response time and enhances predictability.

This article addresses the challenges exposed above by proposing a novel task-to-accelerator mapping for heterogeneous multi-GPU platforms, specially designed to enhance both efficiency and predictability in OpenMP heterogeneous applications, and for which initial ideas were presented in [16]. The main contributions of this work are:

1. A set of lightweight heuristic algorithms for efficient and transparent task-to-accelerator mapping on heterogeneous platforms.
2. A suspension-aware mechanism that increases CPU availability and resolves the busy-waiting problem in synchronous procedures.
3. An evaluation of the proposal on OpenMP-compatible synthetic graphs under different settings.
4. An assessment of the proposal using graphs representative of kernels of real-world applications under different configurations.

The structure of the paper is organized as follows. Section 2 covers the basic concepts of homogeneous and accelerator models in OpenMP. Section 3 discusses some of the existing mapping methods performed on heterogeneous platforms. Section 4 describes the proposed task-to-accelerator mapping in OpenMP. Section 5 evaluates the performance of the proposed mapping in comparison with state-of-the-art works using synthetic graphs and real-world applications with different configurations. Finally, Section 6 concludes the paper and mentions future work.

2. Background

OpenMP is the de facto standard parallel programming model for shared memory architectures in the HPC domain. It provides a parallel API that is portable across several platforms and vendors. OpenMP includes a *thread-centric model* for parallelizing loop-intensive and data-parallel workflows, as well as a *task-centric model* for parallelizing dynamic and irregular applications.

The tasking features enable the application to be represented as a Task Dependency Graph (TDG), a DAG in which nodes are tasks and edges describe data dependencies. This representation is compatible with the traditional representation of real-time systems. This fact has facilitated the interest of the real-time community in OpenMP, which has devoted significant efforts to lean OpenMP towards real-time systems through features to define [13,17] and analyze [14,18] timing constraints and functional correctness [19–21].

OpenMP also includes a host-centric *accelerator model* that relies on tasking for offloading work to accelerators and on extended threading for exploiting parallelism within them. The host is responsible for distributing tasks across resources, including the host and several target devices. A *device* is a logical execution engine that includes a data environment where local memory is stored. Variables are mapped from the host's data environment to the target's.

Fig. 1 shows an OpenMP code snippet with a *host task* and a *target task*, where the second depends on the first, as defined in the *depend* clause through variable *var1*. The target task also defines (a) the data that has to be moved *to* and *from* the device using the *map* clause with the modifiers *to* and *from*, correspondingly, and (b) the target task as asynchronous using the *nowait* clause. The program's execution

starts on the host, with a thread creating two tasks and waiting in the *taskwait* construct. The host task runs on the host, and once it finishes, the target task is offloaded to the device, while the host resource remains idle, ready to execute other tasks if there are any.

If the *nowait* clause is not used, the task becomes synchronous, blocking the host thread while offloading data and computation to the accelerator and waiting for the work to complete. Typical OpenMP implementations, like the LLVM compiler, do not create a task for synchronous offloading, leaving this code invisible to TDG construction. Therefore, in this work, we consider asynchronous target tasks, which are a better option for improving system predictability.

3. Literature review

The amount of work developed in the scope of GPU scheduling highlights the impact of this process on the response time of applications running on heterogeneous platforms. This section describes state-of-the-art methods and motivates the paper's contributions based on their shortcomings.

Elliott and Anderson [5] introduced the Optimal *k*-Exclusion Global Locking Protocol (O-KGLP) for multiprocessor systems, an asymptotically optimal protocol under suspension-oblivious schedulability analysis for synchronous GPU tasks using global scheduling.

Capodici et al. [6] introduced a prototype of a deadline-based scheduler with a preemption policy for GPU tasks running in a virtualized environment. A constant-bandwidth server is used to isolate application tasks within the NVIDIA software stack. It uses an event-driven approach rather than a predefined table-based method for offloading GPU commands.

Kim et al. [3] presented a server-based approach for GPU management to address two issues: busy waiting and long priority inversion. The approach targets multi-core platforms with a single general-purpose GPU (GPGPU) handled through a single lock and the Multiprocessor Priority Ceiling protocol. Tasks are executed via a non-preemptive, sequential process.

Tsog et al. [4] investigated static allocation approaches to mitigate the bottleneck caused by excessive GPU usage in single-GPU systems. This work analyzes the efficiency of balanced CPU/GPU usage in real-time systems, using various heuristics to improve schedulability. Techniques such as Support Vector Machines (SVMs) are proposed but not evaluated due to the high overhead they entail.

Olmedo et al. [8] presented a hierarchical arbitration approach for GPU scheduling on NVIDIA platforms, with the application scheduler at the top of the scheduling hierarchy. The system assumes that CUDA applications can utilize multiple streams, enabling hierarchical scheduling in which streams are allocated in a FIFO queue via a block scheduler that maps blocks to Streaming Multiprocessors (SMs).

Houssam-Eddine et al. [9] presented the HPC-DAG model and a schedulability analysis for heterogeneous systems. The model allows alternative implementations to adapt to different processing units via conditional branches. The mapping process selects the task with the lowest total execution time (TET) from the queue to execute, and the best-fit (BF) or worst-fit (WF) algorithm is used to assign a queue to each task.

Amert et al. [7] introduced the TimeWall framework to provide timing partitions on platforms composed of a multi-core and an accelerator. The framework includes a table-driven scheduler to allocate time slots to partitions, global EDF schedulers within partitions, and a locking protocol to organize accelerator accesses using time slot boundaries.

Yu et al. [10] proposed taskgraph-based execution in OpenMP for GPU devices by implementing an OpenMP taskgraph to device graph transformation on top of the LLVM compiler to exploit NVIDIA platforms through CUDA graphs and AMD platforms through HIP graphs.

The methods mentioned above have the following shortcomings: (i) considering a single GPU device limits scalability [3,4], (ii) some

```

#pragma omp task depend(out: var1)
{ /* host task code */ }
#pragma omp target nowait depend(in: var1) \
    map(to: var2) map(from: var3)
{ /* target task code */ }
#pragma omp taskwait

```

Fig. 1. OpenMP accelerator model example.

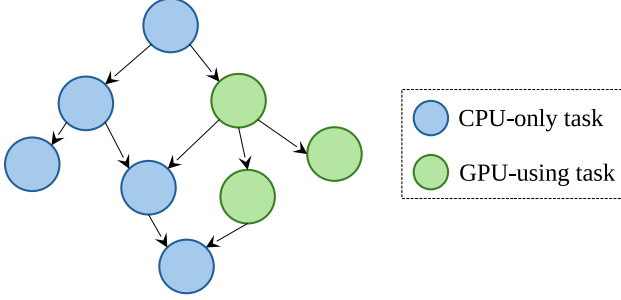


Fig. 2. A random DAG with CPU-only and GPU-using tasks.

algorithms entail high overhead, increasing the mapping overhead and the application response time [4], (iii) a lack of evaluation under a representative variety of configurations limits generalizability [3,5–7], (iv) busy-waiting techniques for executing GPU tasks increase application response time [5,7], and (v) the lack of using isolated algorithms at the different stages of the mapping does not allow understanding their isolated impact in the response time [5,6,8–10]. In light of these shortcomings, this paper proposes an efficient and adaptable task-to-accelerator mapping framework for OpenMP applications running on multi-GPU systems. The system gathers the application's information offline to inform runtime scheduling decisions, reducing overhead and boosting resource utilization.

4. Proposed task-to-accelerator mapping

This section describes the proposed task-to-accelerator mapping method in detail, including the system model, the methodology, the mapping process, and the implementation considerations.

4.1. System model

The system model for this work considers a heterogeneous platform with a multi-core and one or multiple GPUs. OpenMP programs, including tasks and their data dependencies, are defined as DAGs. Host and target tasks are differentiated in the DAG as *CPU-only* and *GPU-using*, respectively. CPU-only tasks run on the CPU, while GPU-using tasks require both the CPU and the GPU. Fig. 2 shows an example of a DAG of a heterogeneous program, where blue tasks are executed in CPUs and green tasks are executed in GPUs.

Fig. 3 illustrates the execution steps of a GPU-using task, which includes (i) two *CPU segments* (blue) for task orchestration (one when the *target* construct is encountered to create the target task and place it in the ready queues when its input dependencies are fulfilled, and another at the end of its execution to synchronize the task and resolve output dependencies that may require the allocation of any new ready tasks), (ii) one *GPU segment* (green) for the kernel execution in the GPU, and (iii) the *data copies* (orange) from the host to the device, occurring before the kernel executes, and from the device to the host, occurring after it has executed.

In this model, GPU-using tasks can be distributed across multiple GPUs using a sequential, non-blocking, and non-preemptive approach. During execution, the CPU segment is suspended while the kernel runs on the GPU, freeing the associated resources. Then, when the GPU completes, it signals the CPU to resume the task. This behavior corresponds to the execution model of OpenMP asynchronous target tasks, which allows the CPU thread executing a target task to continue while it is still waiting for the GPU to complete the computation. Consequently, the work considers some or all parts of a self-suspension segment as available time during which the CPU thread can execute other tasks.

The system model considers applications (i.e., execution scenarios) with an offloading model where offloading to the GPU is a self-contained transaction, and therefore does not consider persistent device-resident data across multiple offloads. This process may overestimate transfer costs, but the goal is to consider the worst-case scenarios. In fact, the focus is on guaranteeing worst-case timing properties, not on optimizing memory management (future work may consider this possibility).

According to the characteristics mentioned above, a task τ_i can be characterized as $\tau_i = (T_i, D_i, C_i, G_i, \xi_i, \sigma_i)$ in the proposed model, where T_i indicates period of each job of the task, D_i indicates relative deadline, C_i represents total execution time of the CPU execution segments, G_i represents execution time of the GPU execution segment, ξ_i denotes memory copy time from the host to the device, and σ_i denotes memory copy time from the device to the host.

The execution time of CPU-only tasks is determined by the execution time of the CPU segment with Eq. (1):

$$Time_i^{CPU} = C_i \quad (1)$$

The execution time of GPU-using tasks is calculated with Eq. (2):

$$Time_i^{GPU} = C_i + \xi_i + G_i + \sigma_i \quad (2)$$

The execution time for the offloading process is determined by Eq. (3):

$$Time_i^{Offloading} = \xi_i + G_i + \sigma_i \quad (3)$$

The work assumes that task's execution time is determined by actual execution during an offline phase. Furthermore, the model described above assumes OpenMP and OS threads are bound one-to-one, which in turn are bound to processing cores. OpenMP provides different thread-affinity policies for core binding, e.g., *close*, which assigns threads with consecutive numbers to the same core or adjacent cores, and *spread*, which creates partitions and distributes successive threads in a round-robin fashion across them. For simplicity, this work considers only single DAG applications in the evaluation results, since many practical parallel workloads (e.g., DAG-based perception pipelines, GPU kernels, or control loops) can be represented as a single recurring DAG. However, multiple DAGs can also be considered if executed sequentially. The work considers that the workload of GPU-using tasks is typically higher than that of CPU-only tasks. Therefore, the overhead of the mapping process and the contention in the queues are low, compared to cases where their workloads are equal or very close.

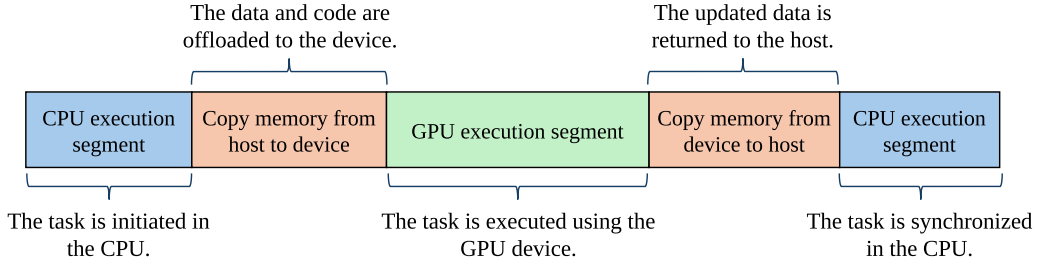


Fig. 3. Execution steps of a GPU-using task.

4.2. Methodology

Fig. 4 illustrates the different components involved in the proposed mapping process, including the *host*, the *accelerators*, and the *offloading manager*, as well as the relationships among them. CPUs include an allocation queue *AQ* for CPU-only tasks and CPU segments of GPU-using tasks. CPUs also have a waiting queue *WQ* to store the second CPU segment of GPU-using tasks, which must be executed after the GPU segment completes in the accelerator. An offloading manager is responsible for orchestrating communication between the CPU and the GPUs. It organizes a global queue *GQ* and multiple local queues *LQs* assigned individually to each GPU, where *GQ* is created by this manager and *LQs* exist in the accelerator (referred to as hardware command queues or work queues). The *GQ* is shared among all *LQs* and is required because it allows the system to make more holistic decisions, and *LQs* have a limited GPU capacity, following a model similar to that considered in some existing works [5,22,23]. The *GQ* has no restricted capacity as it is placed in the CPU. The offloading manager allocates GPU segments to the *GQ* after a *GPU request* is launched from the CPU. Then, tasks are allocated to one of the *LQs* following different scheduling algorithms. The offloading manager runs on the CPU because it is embedded in the OpenMP runtime. Note that the GPUs do not share any memory. Furthermore, only one task can be dispatched from each *LQ* to the corresponding GPU, which indicates a limitation of this work.

The proposed GPU-scheduling method has three main phases: *GQ* selection, *LQ* allocation, and *LQ* dispatching. Three heuristic-based algorithms are suggested for each phase. The following paragraphs describe each phase and the heuristics implemented in it.

4.2.1. *GQ* selection phase

Since *LQs* have a restricted capacity, GPU execution segments are first allocated to the *GQ* and then to the selected *LQ*. A periodic process checks if the *GQ* is not empty and any *LQ* has space to allocate a new task (i.e., GPU segment). In that case, the task is removed from the *GQ* and assigned to the selected *LQ*. The *GQ* queue is ordered based on a given algorithm; therefore, the offloading manager inserts a new task into the appropriate position in a queue, while tasks are selected from the head of the queue. The selection process is performed using the following three possible algorithms based on heuristics or Multi-Criteria Decision Analysis (MCDA):

Least execution time (LET). This algorithm chooses the task in the *GQ* with the shortest execution time. It aims to unleash the maximum available parallelism, since executing the task with minimum execution time can lead to creating outgoing tasks at the next level earlier and, ultimately, reduce the application response time.

Maximum number of all outgoing tasks (MNAOT). This algorithm selects the task with the most significant number of outgoing tasks (i.e., subsequent tasks that have some dependency with the task). It aims, again, to unleash the maximum amount of parallelisms among sibling tasks. The hypothesis is that releasing dependencies will enable more concurrent tasks, thereby potentially reducing end-to-end response time.

Selection weighted-sum model (SWSM). This algorithm chooses the most appropriate task through an MCDA decision based on the execution time (ET) and the Number of All Outgoing Tasks (NAOT) using Eq. (4):

$$SWSM_i = ET_w * ET_i + NAOT_w * \frac{1}{NAOT_i} \quad (4)$$

, where ET_w is the weight assigned to the metric ET, ET_i is the ET of task i , $NAOT_w$ is the weight assigned to the metric NAOT, and $NAOT_i$ is the NAOT of task i . Note that ET_w and $NAOT_w$ are constant values for all tasks of a given DAG, where $ET_w + NAOT_w = 1$. This algorithm aims to leverage both LET and MNAOT, utilizing local and global information of the TDG. Since this algorithm selects the task with the minimum value, and NAOT performs efficiently with the maximum number, NAOT is inverted in the formula.

The execution time of tasks (used in LET) is known through offline analysis of real execution, the information about the descendant tasks (used in MNAOT) is intrinsic to the DAG structure, and the coefficients for each algorithm are set for each execution; therefore, since the *GQ* is ordered (i.e., priority queue), the time complexity of the algorithms is $\mathcal{O}(\log n)$ for insertion and $\mathcal{O}(1)$ for deletion, where n is the number of GPU-using tasks.

4.2.2. *LQ* allocation phase

After selecting a task from the *GQ*, it has to be allocated in an *LQ* with one or more free spaces. Three algorithms are suggested for this process:

Least total execution time (LTET). This algorithm selects the *LQ* with the smallest total execution time. It aims to enhance queue load balancing, resulting in a noticeable reduction in application response time.

Minimum number of jobs (MNJ). This algorithm chooses the *LQ* with the fewest number of tasks. It aims to mitigate LTET's overhead in scenarios where all tasks have similar workloads, while maintaining an acceptable load balance across GPUs.

Allocation weighted-sum model (AWSM). This algorithm chooses the appropriate task in each *LQ*, weighing the number of jobs (NJ) and total execution time (TET) as shown in Eq. (5), and selecting the least value.

$$AWSM_i = TET_w * TET_i + NJ_w * NJ_i \quad (5)$$

, where TET_w is the weight of TET, TET_i is the TET of queue i , NJ_w is the weight of NJ, and NJ_i is the number of tasks in queue i . Note that TET_w and NJ_w are constant values for all queues defined for each DAG, where $TET_w + NJ_w = 1$. This algorithm aims to leverage the benefits of the LTET and MNJ, as both target enhancing queue load balance.

The total execution time of the tasks in each queue (used in LTET) and the number of tasks in each queue (used in MNJ) are updated after each operation, and the coefficients are set for each execution. Therefore, the time complexity of the algorithms is $\mathcal{O}(q)$, where q is the number of GPU devices.

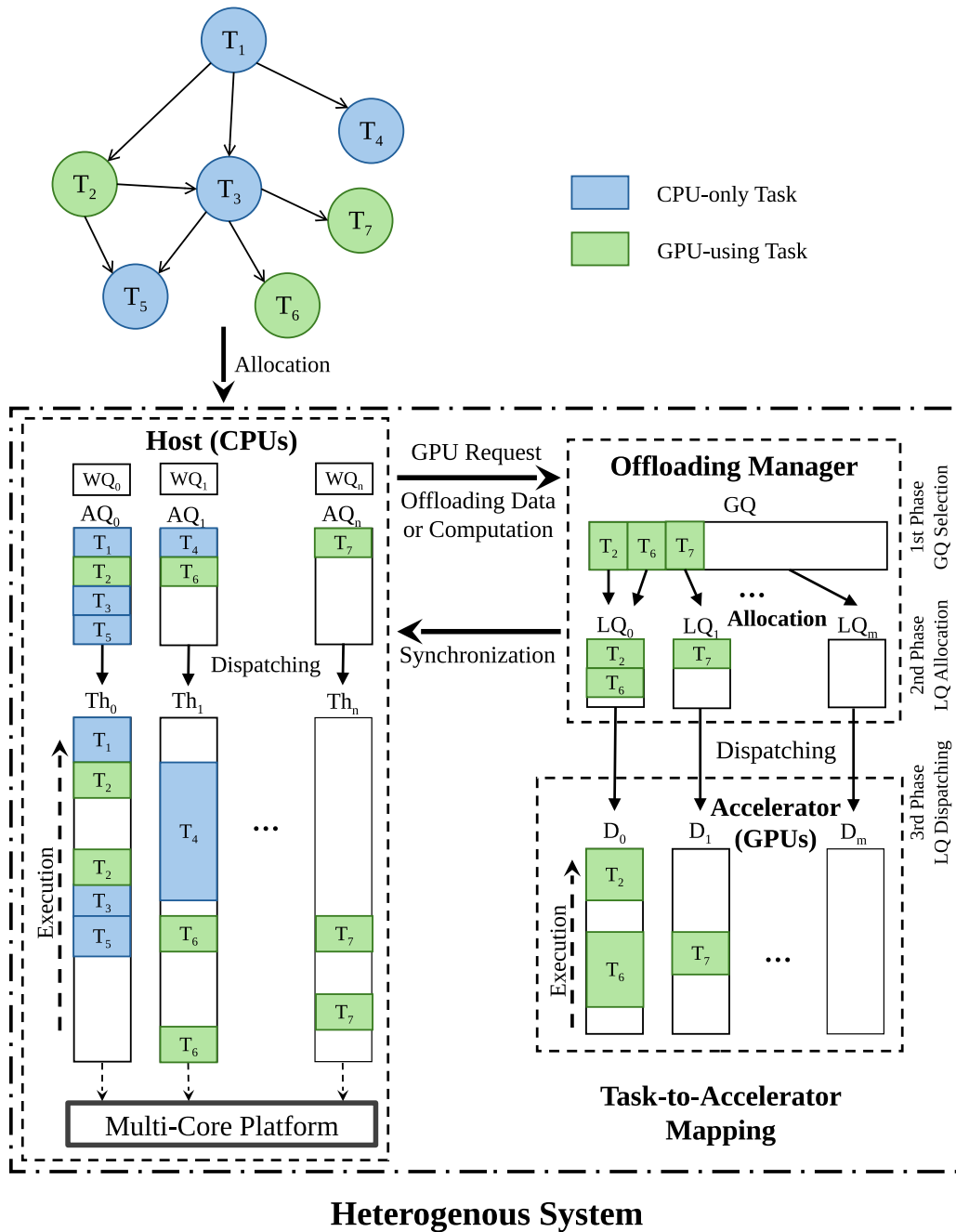


Fig. 4. Proposed task-to-accelerator mapping methodology.

4.2.3. LQ dispatching phase

During this phase, the tasks from the LQs are dispatched to the corresponding GPUs when these are not busy. If there is more than one task in the LQ, one of the tasks will be selected. Since this process is identical to the GQ selection phase, the LQ dispatching algorithms proposed are the same as those used in that phase.

4.3. Mapping process

Algorithm 1 outlines the simulation of the mapping process for a system model described in the previous section, where the simulation algorithm is publicly provided in [24]. It consists of two main procedures to schedule OpenMP tasks:

CPU scheduling. The procedure for CPU scheduling consists of three main phases: (i) *task execution*, where CPU-only and GPU-using tasks are executed using threads, (ii) *task allocation*, where the main thread allocates tasks to queues using a series of heuristics defined in the heuristic task-to-thread mapping [25], and (iii) *task dispatching*, where tasks are dispatched to threads using a series of heuristics also defined in the heuristic mapping.

GPU scheduling. The procedure for GPU scheduling also consists of three major phases: (i) *task execution*, where GPU-using tasks are executed using GPU devices, (ii) *task allocation*, where tasks are allocated to LQs using the proposed heuristics, and (iii) *task dispatching*, where tasks are dispatched to GPU devices using the proposed mapping

Algorithm 1 Task-to-Accelerator Mapping Simulation

Require: N : total number of tasks; N_{CPU} : number of CPU threads; N_{GPU} : number of GPU devices
Ensure: t : total elapsed time steps

```

1: function MAPPING( $N, N_{CPU}, N_{GPU}$ )
2:    $n \leftarrow 0$ ;  $t \leftarrow 0$ 
3:   while  $n < N$  do
4:     // CPU scheduling
5:     for  $i \leftarrow 0$  to  $N_{CPU} - 1$  do
6:       // Execute tasks
7:       if thread  $th_i$  busy then
8:         if task  $t_n$  is CPU-only then
9:           if  $t_n$  completed then  $n \leftarrow n + 1$ 
10:        else if task  $t_n$  is GPU-using then
11:          if  $t_n.ini\_CPU\_seg$  completed then allocate  $t_n$  to  $GQ$ 
12:          else if  $t_n.fin\_CPU\_seg$  completed then  $n \leftarrow n + 1$ 
13:        // Allocate tasks in  $AQs$ 
14:        if thread  $th_i$  is main thread then
15:          for all ready tasks  $t_r$  in DAG do
16:            if  $\exists$  idle thread  $th_j$  with empty queue  $q_j$  then
17:              allocate  $t_r$  to  $q_j$ 
18:            else
19:              select a queue  $q_k$  using a heuristic; allocate  $t_r$ 
20:              to  $q_k$ 
21:          // Dispatch tasks to threads
22:          if thread  $th_i$  idle then
23:            if  $WQ_i$  not empty then
24:              dispatch head task to  $th_i$  (FIFO)
25:            else
26:              select task  $t_r$  from  $AQ_i$  with heuristic; dispatch  $t_r$ 
27:              to  $th_i$ 
28:        // GPU scheduling
29:        for  $j \leftarrow 0$  to  $N_{GPU} - 1$  do
30:          // Execute tasks
31:          if device  $j$  busy then
32:            if kernel execution completed then
33:              allocate the task to  $WQ$  of related CPU thread
34:          // Allocate tasks in  $LQs$ 
35:          for all tasks in  $GQ$  do
36:            select task using GQ selection algorithm; select  $LQ$ 
37:            using LQ allocation algorithm; allocate task to  $LQ$ 
38:          // Dispatch tasks to devices
39:          if device  $D_j$  idle then
40:            select a task  $t_r$  from  $LQ_j$  using heuristic; dispatch  $t_r$ 
41:            to  $D_j$ 
42:         $t \leftarrow t + 1$ 
43:   return  $t$ 

```

method for execution. GPU task execution is non-preemptive, meaning tasks are not suspended once execution begins.

4.4. Implementation considerations

The proposed algorithms can be used with two different approaches: static and dynamic mapping. In static mapping, the simulation process is used to determine the most appropriate mapping of tasks to CPU threads and GPU devices. Afterwards, the program is modified to force the execution of tasks based on the dedicated threads and, finally, it is executed on the platform. Alternatively, instead of modifying the program to force the execution, an approach such as [10] can be used to specify the static mapping. This static mapping may allow for more determinism, important for hard real-time systems.

A dynamic mapping approach may, on the other hand, allow for higher performance, and would require changing the implementation of the underlying scheduler (similar to what was done in [26] for CPU mapping). However, dynamic mapping may present high overload for some of the proposed algorithms, although this can be evaluated based on the required hardware and parallel platform specifications.

5. Performance evaluation

This section evaluates the proposed mapping methodology against O-KGLP [5], TET-BF-TET [9], and TET-WF-TET [9], using the simulation process presented in Algorithm 1. The comparison methods are selected based on their relevance and support for multi-GPU platforms. The evaluation considers synthetic graphs (i.e., random DAGs) compatible with the OpenMP tasking model and graphs representative of three kernels of real-world applications, which are used across different system configurations to provide a comprehensive analysis of the benefits and drawbacks of the proposed method.

5.1. Synthetic graphs

5.1.1. Simulation setup

The simulation process for each configuration setup is executed in multiple iterations. For each iteration, a new random graph is generated, with the task and system characteristics determined randomly. However, the simulation parameters are considered for the generation of DAGs (e.g., generating the task execution time randomly based on the minimum and maximum possible execution times). The GPU workload is at most ϕ times that of CPU-only tasks, assuming the GPU workload is manageable given the offloading overhead. The number of GPU-using tasks is calculated based on the maximum number of tasks and the probability of determining the task type. The task execution time is determined randomly over multiple iterations, varying the minimum, average, and maximum generated numbers for each task. Since this work targets schedulable and predictable parallel applications, the maximum is considered for analysis, as it represents worst-case execution time (WCET), enabling the solution to better meet the timing requirements of each functionality. Each task's deadline is calculated using two specified numbers (DT_{Min} and DT_{Max}) defined in the simulation parameters. The graph deadline is calculated based on the graph volume and two certain numbers (DG_{Min} , DG_{Max}). Furthermore, the response time for each task is calculated based on the graph deadline, the task execution time, and its volume.

Table 1 describes the simulation parameters, where Max_T is the maximum number of tasks, ρ is the probability to define edges (i.e., data dependency) between adjacent vertices, χ is the task type probability (determining the task type), ϕ is the ratio of the workload of GPU-using OpenMP tasks to the workload of CPU-only OpenMP tasks, Itr_T is the number of iterations to determine the execution time of tasks, ET_{Min} is the minimum task execution time, ET_{Max} is the maximum task execution time, ET_{Type} is the type to calculate ultimate task execution time, Itr_G is the number of iterations to execute the graph, N_{CPU} is the number of CPUs, N_{GPU} is the number of GPU devices, and Loc_{Cap} is the capacity of the local queues which is determined based on the former parameters according to the assumption of a relevant work [5]. ET_w , $NAOT_w$, TET_w , and NJ_w are also the coefficients of the $SWSM$ and $AWSM$ algorithms, whose values in the simulations were selected as the best possible values. Both the execution time and the number of all outgoing tasks are important characteristics in selecting tasks from the queues (i.e., GQ and LQs). The former uses local information to increase the number of outgoing tasks at the next level, while the latter uses global information to increase the total number of outgoing tasks. Therefore, the coefficient of the latter can be considered slightly higher than the former to release the maximum available parallelism and reduce the application's response time. Furthermore, both the total execution time and the number of jobs are essential in selecting the

	Coefficient			
	ET_w	$NAOT_w$	TET_w	NJ_w
Case 1	0.1	0.9	0.1	0.9
Case 2	0.4	0.6	0.7	0.3
Case 3	0.5	0.5	0.5	0.5
Case 4	0.9	0.1	0.9	0.1

ρ	0.01					0.03					0.05				
Max_T	100	200	300	400	500	100	200	300	400	500	100	200	300	400	500
Case 1	310	385	489	507	603	354	442	630	684	825	408	661	784	1031	1161
Case 2	252	329	387	443	526	313	428	527	629	735	347	519	721	923	994
Case 3	274	338	415	493	563	323	457	525	638	771	383	540	747	1005	1071
Case 4	279	380	434	545	549	331	488	540	840	883	386	610	855	1022	1122

Fig. 5. Evaluation of the coefficients under different values to identify the values with the most efficient effect on minimizing the WCRT of synthetic graphs, where the response time is considered in *ms*. The green represents a low response time, and the red represents a high response time. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 1

Simulation parameters.

Parameter	Default value
Max_T	1000
ρ	0.01
χ	0.99
ϕ	10
Itr_T	3
ET_{Min}	1 ms
ET_{Max}	50 ms
ET_{Type}	Maximum
DT_{Min}	1
DT_{Max}	5
DG_{Min}	1
DG_{Max}	2
Itr_G	50
N_{CPU}	32
N_{GPU}	8
Loc_{Cap}	$[N_{CPU}/N_{GPU}]$
ET_w	0.4
$NAOT_w$	0.6
TET_w	0.7
NJ_w	0.3

most appropriate *LQs* for ready tasks. However, the former can be considered superior to the latter, as it can improve the load balancing of queues in terms of task execution time, and therefore tasks can be executed in a balanced way and the response time is significantly reduced. In addition, Fig. 5 shows the effect of different values of these coefficients on the response time of synthetic graphs, where $ET_w = 0.1$, $NAOT_w = 0.9$, $TET_w = 0.1$, and $NJ_w = 0.9$ for Case 1, $ET_w = 0.4$, $NAOT_w = 0.6$, $TET_w = 0.7$, and $NJ_w = 0.3$ for Case 2, $ET_w = 0.5$, $NAOT_w = 0.5$, $TET_w = 0.5$, and $NJ_w = 0.5$ for Case 3, as well as $ET_w = 0.9$, $NAOT_w = 0.1$, $TET_w = 0.9$, and $NJ_w = 0.1$ for Case 4, to demonstrate whether the aforementioned justification is reasonable. For each case, different graphs are generated, and the worst-case response time (WCRT) of the response times obtained in multiple iterations is considered in the results. As can be seen in the comparison results, the selected values ($ET_w = 0.4$, $NAOT_w = 0.6$, $TET_w = 0.7$, and $NJ_w = 0.3$ for Case 2) reduce response time in most cases. Note that the simulation parameters are the same for each random DAG generated in each iteration of the simulation process; otherwise, the difference is explicitly mentioned in the results.

The parameters are set to mimic realistic scenarios: high parallelism mimics embarrassingly parallel applications, and reduced parallelism represents applications with a medium to large number of tasks and their data dependencies. Task execution times are similar to state-of-the-art cases in OpenMP applications, ranging from thin to coarse tasks, where the benchmarks considered in this evaluation are publicly

provided in [27]. Furthermore, the number of CPUs and GPUs, respectively, is set to different combination, including: (a) a workstation setup, with {4, 2} CPUs and GPUs, which could contain an Intel Xeon DP Quad-core E5440 CPU and two NVIDIA Quadro NVS 290 GPUs; (b) a server setup, with {8, 4} and {16, 4} CPUs and GPUs, which could contain an AMD EPYC 7302 CPU and 4 NVIDIA RTX 2080 GPUs; and (c) a supercomputing node setup, with {32, 8} and {64, 8} CPUs and GPUs, which could be a node of the Marenostrum ACC supercomputer. Finally, the simulations are executed on a 12th Gen Intel(R) Core(TM) i7-12650H @ 2.30 GHz. However, this does not affect the results, as the execution of the tasks is based on simulations, and not based on real executions on the platform.

5.1.2. Simulation results

The evaluation was conducted over the Cartesian product of all possible combinations of heuristics, although this paper presents only one of the combinations (i.e., LET-LTET-MNAOT) due to space limitations. Fig. 6 illustrates an evaluation of the proposed mapping algorithms under different values of ρ and Max_T . The comparison results shows that LET-LTET-MNAOT and LET-AWSM-MNAOT obtained the lowest WCRT among the other algorithms in most cases. Another achievement is that LET-X-X and SWSM-X-X perform better than MNAOT-X-X because LET and SWSM consider execution time in the selection of tasks from *GQ*. Consequently, we selected LET-LTET-MNAOT as one of the most efficient mapping algorithm for the evaluation process.

The evaluation also considers that the mapping of CPU tasks-to-threads uses the heuristic task-to-thread mapping [25], with tasks assigned to the queues with minimum total execution time (the MTET heuristic) and dispatched by minimum execution time (the MET heuristic). The reason is that this combination primarily focuses on task execution time, making it very useful for achieving a minimum response time.

The results are determined by the DAG's density, which reflects the degree of node connectivity and the amount of parallelism exposed in the application. The density is divided into three classes: high parallelism ('H'), medium parallelism ('M'), and low parallelism ('L'). These configurations can create suitable random DAGs for evaluating the proposed mapping methodology. Fig. 7 shows the assessment of the mapping methods under different conditions in terms of application response time.

Fig. 7(a) to 7(c) illustrate the results for graphs with high parallelism, showing that the worst-case response time and variability obtained with the proposed method are lower than those obtained with the state-of-the-art. The reasons are that the proposed algorithms (LET-LTET-MNAOT) (i) select the task with the shortest execution time from the *GQ*, which enables the execution of the dependant task(s) earlier and improves the work-conserving nature of the mapping process, (ii) select local queues with the shortest total execution time, which to

ρ	0.01					0.03					0.05				
Max _T	100	200	300	400	500	100	200	300	400	500	100	200	300	400	500
LET-LTET-LET	242	368	444	600	649	357	480	601	778	938	387	576	776	918	892
LET-LTET-MNAOT	242	351	393	473	543	357	480	539	646	875	387	576	776	918	892
LET-LTET-SWSM	242	368	444	600	649	357	480	601	778	938	387	576	776	918	892
LET-MNJ-LET	242	368	444	600	658	357	480	601	741	938	387	576	776	918	905
LET-MNJ-MNAOT	242	351	393	525	586	357	480	539	646	875	387	576	776	918	892
LET-MNJ-SWSM	242	368	444	600	658	357	480	601	741	938	387	576	776	918	905
LET-AWSM-LET	242	368	444	600	649	357	480	601	778	938	387	576	776	918	892
LET-AWSM-MNAOT	242	351	393	473	543	357	480	539	646	875	387	576	776	918	892
LET-AWSM-SWSM	242	368	444	600	649	357	480	601	778	938	387	576	776	918	892
MNAOT-LTET-LET	262	386	428	595	647	366	495	603	755	885	394	600	799	955	984
MNAOT-LTET-MNAOT	262	382	408	478	588	366	495	607	681	885	394	600	799	955	984
MNAOT-LTET-SWSM	262	386	428	595	647	366	495	603	746	885	394	600	799	955	984
MNAOT-MNJ-LET	262	373	467	653	641	366	483	620	743	864	394	631	799	955	985
MNAOT-MNJ-MNAOT	262	369	406	514	569	366	483	620	704	887	394	600	799	955	988
MNAOT-MNJ-SWSM	262	373	467	653	641	366	483	620	736	864	394	631	799	955	985
MNAOT-AWSM-LET	262	386	428	595	647	366	495	603	755	885	394	600	799	955	984
MNAOT-AWSM-MNAOT	262	382	408	478	588	366	495	607	681	885	394	600	799	955	984
MNAOT-AWSM-SWSM	262	386	428	595	647	366	495	603	746	885	394	600	799	955	984
SWSM-LTET-LET	242	355	444	645	654	357	480	602	702	917	387	576	776	918	943
SWSM-LTET-MNAOT	242	351	399	476	556	357	480	541	693	939	387	576	776	918	943
SWSM-LTET-SWSM	242	355	444	645	649	357	480	602	702	917	387	576	776	918	943
SWSM-MNJ-LET	242	355	444	600	695	357	480	602	735	933	387	576	776	918	945
SWSM-MNJ-MNAOT	242	351	399	525	586	357	480	541	646	891	387	576	776	918	992
SWSM-MNJ-SWSM	242	355	444	600	653	357	480	602	735	933	387	576	776	918	945
SWSM-AWSM-LET	242	355	444	645	654	357	480	602	702	917	387	576	776	918	943
SWSM-AWSM-MNAOT	242	351	399	476	556	357	480	541	693	939	387	576	776	918	943
SWSM-AWSM-SWSM	242	355	444	645	649	357	480	602	702	917	387	576	776	918	943

Fig. 6. Evaluation of the proposed mapping algorithms in terms of obtaining the WCRT of synthetic graphs under different values of ρ and Max_T , where the response time is represented in *ms*. The green indicates a low response time, and the red indicates a high response time. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

improve the load balancing of the system, and (iii) select tasks with the largest number of all outgoing tasks from the *LQs*, leading to an increase in the number of ready tasks. Furthermore, the proposed method has a greater impact as the number of computing elements increases, because this configuration exposes a large number of parallel tasks, enabling better selection of the most suitable local queues and tasks.

Fig. 7(d) to 7(f) show similar improvements for the proposed method. In cases with more data dependencies among tasks, a more accurate selection of queues and tasks may have a greater impact than in embarrassingly parallel cases. The baseline and TET-BF-TET methods exhibit excessively penalized average response time and variability. In contrast, the proposed method shows stability in these values, particularly relevant in configurations with fewer resources. The reasons are that (i) the baseline selects tasks from queues in a FIFO manner, reducing the work balance across the queues and the system's ability to maintain a work-conserving nature, and (ii) the TET-BF-TET method uses the best-fit (BF) algorithm, which can also reduce load balancing and significantly increase response time.

Finally, Fig. 7(g) to 7(i) show improvements similar to the previous case. In this case, the proposed method is comparable to O-KGLP and TET-WF-TET, the state-of-the-art algorithms, when using the maximum available resources in the proposed configurations, and minimally enhances them when fewer resources are available. The reason is that the number of parallel tasks is low in this configuration because the number of data dependencies is high. Therefore, the impact of the algorithm used to assign a large number of resources to a small number of tasks is minimal.

Overall, the evaluation shows that the proposed heuristics reduce execution time and the application's response time variability in the best-case scenario, when there are enough parallel tasks to make a difference in task ordering and distribution. Furthermore, the results confirm that the proposed method does not penalize execution when

the parallel nature of the application is limited or there are not enough resources to make a difference in task distribution.

5.2. Real-world applications

5.2.1. Evaluation setup

Applications. The evaluation uses graphs representatives of kernels of three real-world applications, including *Axpy*, *Dotp*, and *Heat*, from the HPC domain. *Axpy* contains an embarrassingly parallel program, performing a basic linear algebra operation, where the program multiplies the vector *X* by the scalar *A* and adds the vector *Y*. The configuration generates 109 independent OpenMP tasks, including 10 CPU-only tasks and 99 GPU-using tasks. *Dotp* also involves an embarrassingly parallel program, creating a scalar by multiplying two vectors. Its program generates 155 independent OpenMP tasks, containing 5 CPU-only tasks and 150 GPU-using tasks. These applications are appropriate for real-world use cases where there are no data dependencies between tasks, enabling a large number of parallel executions to assess the performance of the mapping. Note that OpenMP tasks are assigned to queues at the beginning of the task mapping in these programs. *Heat* performs a heat diffusion simulation implemented through the Gauss-Seidel method, demonstrating a stencil computation. It is best suited for scenarios where the number of parallel executing tasks is moderate, enabling for mapping evaluation using a few OpenMP tasks. Two configurations are considered for this application: (i) the program generates 128 OpenMP tasks, where all the tasks are GPU-using tasks, with data dependencies between them, and (ii) it generates 220 OpenMP tasks, including 80 CPU-only tasks and 140 GPU-using tasks, with data dependencies. It is worth noting that more details about these programs can be found in [24] (the 'benchmark' folder).

Hardware and Operating System. An NVIDIA Jetson AGX Xavier with an 8-core NVIDIA Carmel Arm v8.2 64-bit CPU and 32 GB 256-bit

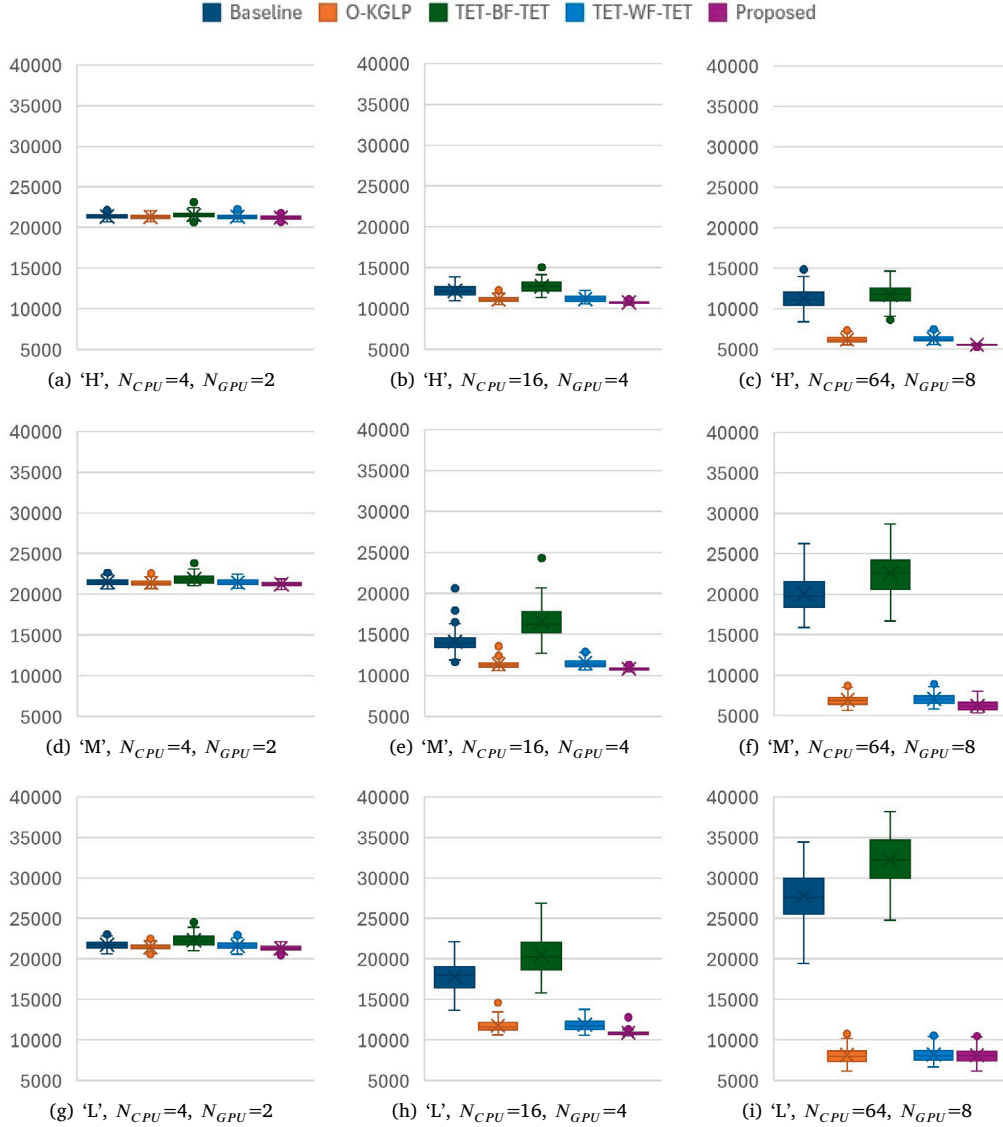


Fig. 7. Evaluation results for different DAGs, where each line represents a different graph density, each column represents a different hardware setup, the x-axis represents the different mapping methods, and the y-axis indicates the response time (ms).

RAM is used to execute the programs and perform tracing using the profiling tools. The same platform mentioned at the end of Section 5.1.1 is utilized to execute the simulations.

Compiler and Runtime System. An extended version of LLVM/Clang 17.0.0, provided by the Barcelona Supercomputing Center (BSC) [28], is used to execute the programs on the platform. It supports OpenMP 5.0 and the extensions for materializing a TDG, known as *Taskgraph*.

Profiling Tools. The Extrae instrumentation toolkit [29] is applied to perform the host-side (i.e., CPU) tracing for measuring task execution time. A script [30] is also used to generate (in the first iteration) or update (in the subsequent iterations) a JSON file based on the tracing information, where this file contains the TDG structure and the execution time of all tasks. Furthermore, a CUPTI-based activity tracing toolkit [31] is used to perform tracing on both the host side (i.e., CPU) and the device side (i.e., GPU) for measuring execution time of the GPU execution segment, memory copy time from the host to the device, and memory copy time from the device to the host. Note that the executions are done in multiple iterations to calculate the WCRT for each aforementioned attribute, since the work is done for hard real-time tasks.

Configuration. The programs are executed on the platform in 10 iterations to measure the execution time of tasks on the host and the device. In addition, most of the parameters, except those used for random DAGs, are considered based on the values shown in Table 1. The graphs of the three kernels, together with the values extracted through tracing, are then used in the same simulation environment as the synthetic graphs.

5.2.2. Evaluation results

Fig. 8 shows the evaluation of proposed mapping (all compositions of the algorithms), compared to the literature works, using *Axpy* and *Dotp*, where different hardware configurations (i.e., computing resources) are considered. The comparison results obtained for *Axpy* indicate that (i) most of the algorithms perform efficiently in most cases, (ii) LET-X-X and SWSM-X-X work better than MNAOT-X-X, especially when the amount of hardware resources is low or medium, as the task execution time is not low in this application and, therefore, the algorithms that consider execution time in the GQ selection phase perform well by improving the work conservation of the mapping, and (iii) MNAOT-X-X works better than the other sets in some cases

N_{CPU}, N_{GPU}	Axy (num_tasks = 109)			Dotp (num_tasks = 155)		
	8, 4	16, 4	32, 8	8, 4	16, 4	32, 8
Baseline	49.70876	49.11773	26.75326	44.714827	44.406787	23.697888
O-KGLP	49.70783	49.18778	26.30887	44.752739	44.8702	23.256691
TET-BF-TET	48.57896	48.12031	25.64134	44.582951	44.572147	22.904986
TET-WF-TET	48.57653	48.07965	25.6247	44.582	44.559589	22.892224
LET-LTET-LET	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
LET-LTET-MNAOT	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
LET-LTET-SWSM	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
LET-MNJ-LET	48.57653	48.07965	25.6247	44.582	44.559589	22.892224
LET-MNJ-MNAOT	48.57653	48.07965	25.62208	44.582	44.559589	22.892224
LET-MNJ-SWSM	48.57653	48.07965	25.6247	44.582	44.559589	22.892224
LET-AWSM-LET	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
LET-AWSM-MNAOT	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
LET-AWSM-SWSM	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
MNAOT-LTET-LET	49.29977	48.79234	24.93202	44.338625	44.523163	22.927352
MNAOT-LTET-MNAOT	49.7098	49.11784	26.72981	44.714874	44.390645	23.505478
MNAOT-LTET-SWSM	49.29977	48.79234	24.93202	44.338625	44.523163	22.927352
MNAOT-MNJ-LET	49.29995	48.79264	25.99829	44.338625	44.523134	22.927352
MNAOT-MNJ-MNAOT	49.71047	49.11814	26.72985	44.714874	44.390637	23.505478
MNAOT-MNJ-SWSM	49.29995	48.79264	25.99829	44.338625	44.523134	22.927352
MNAOT-AWSM-LET	49.29977	48.79234	24.93202	44.338625	44.523163	22.927352
MNAOT-AWSM-MNAOT	49.7098	49.11784	26.72981	44.714874	44.390645	23.505478
MNAOT-AWSM-SWSM	49.29977	48.79234	24.93202	44.338625	44.523163	22.927352
SWSM-LTET-LET	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
SWSM-LTET-MNAOT	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
SWSM-LTET-SWSM	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
SWSM-MNJ-LET	48.57653	48.07965	25.6247	44.582	44.559589	22.892224
SWSM-MNJ-MNAOT	48.57653	48.07965	25.62208	44.582	44.559589	22.892224
SWSM-MNJ-SWSM	48.57653	48.07965	25.6247	44.582	44.559589	22.892224
SWSM-AWSM-LET	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
SWSM-AWSM-MNAOT	48.57644	48.08009	25.62136	44.582	44.559589	22.892216
SWSM-AWSM-SWSM	48.57644	48.08009	25.62136	44.582	44.559589	22.892216

Fig. 8. Evaluation of the proposed mapping algorithms using the *Axy* and *Dotp* applications in different hardware configurations. Response time is considered in s. The green indicates a low response time, while the red indicates a high response time. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

when the amount of resources is high, as many resources execute tasks simultaneously and therefore the response time is reduced.

Furthermore, the evaluation results obtained for *Dotp* show that (i) the proposed algorithms perform almost better than those in the literature, (ii) MNAOT-X-X works better than the others when the amount of resources is low or medium, reducing response time, as the task execution time is not high in this application and the algorithms considering the number of all outgoing tasks for the GQ task selection process can increase the number of next ready tasks and thereby boost the task execution process, (iii) LET-X-X and SWSM-X-X perform more efficiently than MNAOT-X-X when the amount of resources is high, as the work conservation of the mapping is improved across many resources when the shortest execution time is considered in the mapping algorithm for selecting tasks from GQ, and (iv) in these sets, LTET and AWSM also reduce the application's response time, compared to MNJ, by enhancing the load balancing of the queues in terms of task execution time.

Fig. 9 illustrates the performance of the new algorithms using the *Heat* application in two configurations executing with different amounts of computing resources. The comparison results indicate that (i) in the first configuration with a medium number of tasks, LET-X-X and SWSM-X-X perform better in most cases when the amount of resources is low or medium, while MNAOT-X works better when the

amount of resources is high, as a work-conserving mapping is achieved by considering the task execution time and the number of jobs, respectively, and (ii) in the second configuration with a large number of (GPU-using) tasks, MNAOT-X-X mostly works better, improving the work conservation of the mapping by selecting tasks from GQ based on the total number of outgoing tasks.

In summary, LET-X-X and SWSM-X-X perform well in most cases regarding the reduction of response time for the applications with a medium number of tasks. In contrast, MNAOT-X works better in most cases for the applications including a large number of tasks.

6. Conclusion

This paper proposes a heuristic-based task-to-accelerator mapping for OpenMP applications running on heterogeneous platforms. The process aims to reduce the response time (WCRT) and its variability in parallel task-based applications, thereby adapting the OpenMP implementation to the requirements of real-time systems. Accordingly, only asynchronous tasks are considered, preserving the work-conserving nature of threads to reduce the impact on system predictability.

In brief, the mapping process starts when a GPU-using task (*target* task in OpenMP) is encountered and the *offloading manager* allocates it to the *global queue*. Then, the task is allocated to the *local queue* of a

N_{CPU}, N_{GPU}	Heat (num_tasks = 128)			Heat (num_tasks = 220)		
	8, 4	16, 4	32, 8	4, 2	8, 4	16, 4
Baseline	513.97	705.32	705.32	618.01	620.22	858.57
O-KGLP	435.82	435.82	405.97	594.73	485.01	485.01
TET-BF-TET	564.34	1180.6	1180.6	709.28	709.28	1090.42
TET-WF-TET	433.1	433.1	405.98	615.62	498.15	498.15
LET-LTET-LET	433.44	433.44	405.98	597.07	498.15	498.15
LET-LTET-MNAOT	435.61	435.61	405.98	589.78	498.15	498.15
LET-LTET-SWSM	433.44	433.44	405.98	597.07	498.15	498.15
LET-MNJ-LET	433.1	433.1	405.98	615.62	498.15	498.15
LET-MNJ-MNAOT	442.68	442.68	405.98	602.25	498.15	498.15
LET-MNJ-SWSM	433.1	433.1	405.98	615.62	498.15	498.15
LET-AWSM-LET	433.44	433.44	405.98	597.07	498.15	498.15
LET-AWSM-MNAOT	435.61	435.61	405.98	589.78	498.15	498.15
LET-AWSM-SWSM	433.44	433.44	405.98	597.07	498.15	498.15
MNAOT-LTET-LET	438.65	438.65	405.97	590.83	478.32	478.32
MNAOT-LTET-MNAOT	435.6	435.6	405.97	604.18	478.32	478.32
MNAOT-LTET-SWSM	438.65	438.65	405.97	590.83	478.32	478.32
MNAOT-MNJ-LET	433.09	433.09	405.97	592.86	478.32	478.32
MNAOT-MNJ-MNAOT	442.67	442.67	405.97	595.56	478.32	478.32
MNAOT-MNJ-SWSM	433.09	433.09	405.97	592.86	478.32	478.32
MNAOT-AWSM-LET	438.65	438.65	405.97	590.83	478.32	478.32
MNAOT-AWSM-MNAOT	435.6	435.6	405.97	604.18	478.32	478.32
MNAOT-AWSM-SWSM	438.65	438.65	405.97	590.83	478.32	478.32
SWSM-LTET-LET	433.44	433.44	405.98	597.07	498.15	498.15
SWSM-LTET-MNAOT	435.61	435.61	405.98	589.78	498.15	498.15
SWSM-LTET-SWSM	433.44	433.44	405.98	597.07	498.15	498.15
SWSM-MNJ-LET	433.1	433.1	405.98	615.62	498.15	498.15
SWSM-MNJ-MNAOT	442.68	442.68	405.98	602.25	498.15	498.15
SWSM-MNJ-SWSM	433.1	433.1	405.98	615.62	498.15	498.15
SWSM-AWSM-LET	433.44	433.44	405.98	597.07	498.15	498.15
SWSM-AWSM-MNAOT	435.61	435.61	405.98	589.78	498.15	498.15
SWSM-AWSM-SWSM	433.44	433.44	405.98	597.07	498.15	498.15

Fig. 9. Analysis of the proposed mapping algorithms using the *Heat* application under two configurations executing with different computing resources. Response time is shown in s. The green highlights a low response time, while the red highlights a high response time. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

device and executed when the assigned GPU becomes available. These three phases are carried out using different heuristics and Multi-Criteria Decision Analysis (MCDA) techniques. Additionally, the GPU segments are executed asynchronously, allowing the system to overlap GPU and CPU computations without blocking threads in busy-waiting loops.

The performance of the proposed mapping method is evaluated through simulations based on various synthetic graphs representing realistic scenarios and on three kernels of real-world applications with different configurations, including low-to-high degrees of parallelism, small-to-large graphs, and small-to-high numbers of resources, including CPUs and GPUs.

The simulation results through synthetic graphs show that the proposed method achieves better worst-case response time (WCRT) and variability than the most relevant related works, particularly when the number of resources is medium or high, as well as the applications exhibit high or medium parallelism. Furthermore, the evaluation results using three kernels of real-world applications demonstrate that the proposed algorithms significantly reduce application response time by improving the work-conserving mapping and the load balancing of queues through the specifications employed, in the applications with different configurations and various amounts of computing resources.

Future work will (i) evaluate the performance of the proposed mapping using experimental results obtained for real-world OpenMP

applications, running on different hardware platforms from the embedded and high-performance computing domains, including supercomputing nodes with large computing resources, and (ii) consider the execution of tasks using multiple GPUs in the same transaction, addressing memory management optimization.

CRediT authorship contribution statement

Mohammad Samadi: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Tiago Carvalho:** Writing – review & editing, Validation, Methodology, Conceptualization. **Luís Miguel Pinho:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization. **Sara Royuela:** Writing – review & editing, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This research work was performed in the scope of Mohammad Samadi's PhD thesis at Universitat Politècnica de Catalunya, Barcelona, Spain. It has been partially funded by the Chips Joint Undertaking through the MATISSE project (grant agreement No. 101140216) and the HiPART project, with reference PID2023-148117NA-I00, financed by MICIU/AEI /10.13039/501100011033 and FEDER, UE.

Data availability

The data supporting the findings of this study are publicly available at <https://github.com/m-samadi/task-to-accelerator-mapping/>.

References

- [1] R. Leisha, K.J. Medows, M.M. Thiruthuvanantham, S.R. Babu, P. Divakaran, V.M. Chaturvedi, Advanced cyber-physical systems utilizing deep learning for crowd density detection and public safety, in: *Navigating Cyber-Physical Systems with Cutting-Edge Technologies*, IGI Global Scientific Publishing, 2025, pp. 83–122, <http://dx.doi.org/10.4018/979-8-3693-5728-6.ch004>.
- [2] I. Gomez, U.D. de Cerio, J. Parra, J.M. Rivas, J.J. Gutierrez, M.G. Harbour, Using MAST for modeling and response-time analysis of real-time applications with GPUs, *J. Syst. Archit.* 157 (2024) 103300, <http://dx.doi.org/10.1016/j.sysarc.2024.103300>.
- [3] H. Kim, P. Patel, S. Wang, R.R. Rajkumar, A server-based approach for predictable GPU access with improved analysis, *J. Syst. Archit.* 88 (2018) 97–109, <http://dx.doi.org/10.1016/j.sysarc.2018.05.003>.
- [4] N. Tsog, M. Becker, F. Bruhn, M. Behnam, 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, 1, IEEE, 2019, pp. 4516–4522, <http://dx.doi.org/10.1109/IECON.2019.8926767>.
- [5] G.A. Elliott, J.H. Anderson, An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems, *Real-Time Syst.* 49 (2) (2013) 140–170, <http://dx.doi.org/10.1007/s11241-012-9170-0>.
- [6] N. Capodiceci, R. Cavicchioli, M. Bertogna, A. Paramakuru, Deadline-based scheduling for GPU with preemption support, in: *IEEE Real-Time Systems Symposium, RTSS, IEEE*, 2018, pp. 119–130, <http://dx.doi.org/10.1109/RTSS.2018.00021>.
- [7] T. Amert, Z. Tong, S. Voronov, J. Bakita, F.D. Smith, J.H. Anderson, Timewall: Enabling time partitioning for real-time multicore+ accelerator platforms, in: *IEEE Real-Time Systems Symposium, RTSS, IEEE*, 2021, pp. 455–468, <http://dx.doi.org/10.1109/RTSS52674.2021.00048>.
- [8] I.S. Olmedo, N. Capodiceci, J.L. Martinez, A. Marongiu, M. Bertogna, Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective, in: *IEEE Real-Time and Embedded Technology and Applications Symposium, IEEE*, 2020, pp. 213–225, <http://dx.doi.org/10.1109/RTAS48715.2020.000-5>.
- [9] Z. Houssam-Eddine, N. Capodiceci, R. Cavicchioli, G. Lipari, M. Bertogna, The HPC-dag task model for heterogeneous real-time systems, *IEEE Trans. Comput.* 70 (10) (2020) 1747–1761, <http://dx.doi.org/10.1109/TC.2020.3023169>.
- [10] C. Yu, S. Royuela, E. Quiñones, Enhancing heterogeneous computing through openmp and GPU graph, in: 53rd International Conference on Parallel Processing, Association for Computing Machinery (ACM), 2024, pp. 534–543, <http://dx.doi.org/10.1145/3673038.3673050>.
- [11] F. Bonanno, G. Capizzi, G.L. Sciuto, C. Napoli, G. Pappalardo, E. Tramontana, A cascade neural network architecture investigating surface plasmon polaritons propagation for thin metals in openmp, in: *International Conference on Artificial Intelligence and Soft Computing*, Springer, 2014, pp. 22–33, http://dx.doi.org/10.1007/978-3-319-07173-2_3.
- [12] D.A. Padilla, R.A.I. Pajes, J.T. De Guzman, Detection of corn leaf diseases using convolutional neural network with openmp implementation, in: 2020 IEEE 12th International Conference on Humanoid, Nanotechnology, Information Technology, Communication and Control, Environment, and Management, HNICEM, IEEE, 2020, pp. 1–6, <http://dx.doi.org/10.1109/HNICEM51456.2020.9400004>.
- [13] M.A. Serrano, S. Royuela, E. Quiñones, Towards an openmp specification for critical real-time systems, in: *International Workshop on OpenMP*, Springer, 2018, pp. 143–159, http://dx.doi.org/10.1007/978-3-319-98521-3_10.
- [14] J. Sun, N. Guan, X. Wang, C. Jin, Y. Chi, Real-time scheduling and analysis of synchronous openmp task systems with tied tasks, in: 56th Annual Design Automation Conference 2019, 2019, pp. 1–6, <http://dx.doi.org/10.1145/3316781.3317891>.
- [15] V. Kale, W. Lu, A. Curtis, A.M. Malik, B. Chapman, O. Hernandez, Toward supporting multi-GPU targets via taskloop and user-defined schedules, in: *International Workshop on OpenMP*, Springer, 2020, pp. 295–309, http://dx.doi.org/10.1007/978-3-030-58144-2_19.
- [16] M. Samadi, T. Carvalho, L. Miguel Pinho, S. Royuela, Task-to-accelerator mapping for heterogeneous systems using heuristics, *Work-in-Progress Sess. 29th Ada-Europe International Conf. Reliab. Softw. Technol. (AEiC 2025)*, ACM SIGAda Ada Lett. 45 (2) (2025) 44–48, <http://dx.doi.org/10.1145/3821489.3821491>.
- [17] B. McDonald, F. Mueller, OpenMP-RT: Native pragma support for real-time tasks and synchronization with LLVM under linux, in: *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2024, pp. 119–130, <http://dx.doi.org/10.1145/3652032.3657574>.
- [18] M.A. Serrano, E. Quiñones, Response-time analysis of DAG tasks supporting heterogeneous computing, in: *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6, <http://dx.doi.org/10.1145/3195970.3196104>.
- [19] S. Royuela, A. Duran, M.A. Serrano, E. Quiñones, X. Martorell, A functional safety openmp for critical real-time embedded systems, in: *International Workshop on OpenMP*, Springer, 2017, pp. 231–245, http://dx.doi.org/10.1007/978-3-319-65578-9_16.
- [20] T. Cramer, S. Schwitanski, F. Münchhelfen, C. Terboven, M.S. Müller, An openmp epoch model for correctness checking, in: 2016 45th International Conference on Parallel Processing Workshops, ICPPW, IEEE, 2016, pp. 299–308, <http://dx.doi.org/10.1109/ICPPW.2016.51>.
- [21] F. Jin, A. Tao, L. Yu, V. Sarkar, Visualizing correctness issues in openmp programs, in: *International Workshop on OpenMP*, Springer, 2024, pp. 161–175, http://dx.doi.org/10.1007/978-3-031-72567-8_11.
- [22] G.A. Elliott, B.C. Ward, J.H. Anderson, GpUSync: A framework for real-time GPU management, in: *IEEE 34th Real-Time Systems Symposium, IEEE*, 2013, pp. 33–44, <http://dx.doi.org/10.1109/RTSS.2013.12>.
- [23] T. Amert, N. Otterness, M. Yang, J.H. Anderson, F.D. Smith, GPU scheduling on the nvidia TX2: Hidden details revealed, in: *IEEE Real-Time Systems Symposium, RTSS, IEEE*, 2017, pp. 104–115, <http://dx.doi.org/10.1109/RTSS.2017.00017>.
- [24] M. Samadi, The task-to-accelerator mapping simulator, 2025, URL <https://github.com/m-samadi/task-to-accelerator-mapping/>.
- [25] M. Samadi, S. Royuela, L.M. Pinho, T. Carvalho, E. Quiñones, Time-predictable task-to-thread mapping in multi-core processors, *J. Syst. Archit.* 148 (2024) 103068, <http://dx.doi.org/10.1016/j.sysarc.2024.103068>.
- [26] M. Samadi, T. Carvalho, L.M. Pinho, S. Royuela, Evaluation of heuristic task-to-thread mapping using static and dynamic approaches, in: *Workshop on Job Scheduling Strategies for Parallel Processing*, Springer, 2024, pp. 103–119, http://dx.doi.org/10.1007/978-3-031-74430-3_6.
- [27] M. Samadi, Openmp-benchmarks, 2025, URL <https://github.com/m-samadi/OpenMP-benchmarks/>.
- [28] BSC LLVM, 2024, URL <https://gitlab.bsc.es/ampere-sw/wp2/llvm/>.
- [29] Extrae, BSC performance tools, 2024, URL <https://tools.bsc.es/extrae>.
- [30] TDG instrumentation, 2024, URL <https://gitlab.bsc.es/ampere-sw/wp2/tdg-instrumentation-script/>.
- [31] T. Carvalho, L.M. Pinho, Tracing and measuring GPU execution in automotive software systems, *Ada User J.* 43 (3) (2022) 160–164.