Future Generation Computer Systems 176 (2026) 108145

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Solving the task scheduling and GPU reconfiguration problem on MIG devices via deep reinforcement learning

Jorge Villarrubia *, Luis Costero, Francisco D. Igual, Katzalin Olcoz

Universidad Complutense de Madrid, Departamento de Arquitectura de Computadores y Automática, Madrid, 28040, Spain

ARTICLE INFO

Keywords:
Multi-Instance GPU (MIG)
Moldable resource management
Deep reinforcement learning
Task scheduling

ABSTRACT

Recent advances in dynamic GPU partitioning, such as NVIDIA's Multi-Instance GPU (MIG) technology, have enhanced resource utilization by enabling task co-execution without contention. However, existing MIG schedulers remain limited to static or task-agnostic methods that sacrifice optimality for tractability. This paper presents a Deep Reinforcement Learning framework that seeks to minimize the completion time of a task queue by holistically addressing the dimensions of the problem: task molding, GPU reconfiguration and execution order. To manage the vast solution space, we apply optimizations such as discrete and canonical representation of states, unification of equivalent configurations, action masking, or promoting the exploration of reconfigurations; this offers insights for similar resource management scenarios. The proposed models are extensively evaluated with widely used benchmarks of the Rodinia and Altis suites, and synthetic workloads generated to emulate a wide range of plausible real situations. The final model improves to the state-of-the-art, especially in workloads that clearly contradict the assumptions of previous proposals, achieving a difference of less than 20% to the optimum. Additionally, two different approaches to the problem are faced (offline vs. online), discussing their theoretical advantages and disadvantages, and evaluating them experimentally for the final model.

1. Introduction

Recent advances in Artificial Intelligence (AI), mainly driven by the development of Large Language Models (LLMs), have dramatically increased the demand for massively parallel computing. Modern datacenter GPUs, featuring thousands of cores and hundreds of gigabytes of memory, provide PetaFLOPS for LLM training. Many common GPU workloads, however, still underutilize these resources, as evidenced by HPC benchmarks [1,2] and other Deep Learning (DL) tasks [3,4] . This underutilization not only delays task completion, but also wastes a lot of energy due to the huge power consumption of these devices [5] .

A natural solution is to run independent applications concurrently on the GPU to share its resources. NVIDIA supports this with technologies like the well-established Multi-Process Service (MPS) [6] and more recently the Multi-Instance GPU (MIG) [7] technology. MPS allocates certain resources per process, such as Stream Multiprocessors (SMs), while sharing others, such as memory bandwidth, which can lead to contention and quality-of-service issues [1,8]. In contrast, MIG partitions the GPU into isolated virtual instances with dedicated resources (SMs, L2 cache, DRAM and memory channels), ensuring predictable performance [8,9]. Moreover, MIG supports flexible and dynamic reconfiguration, allowing one virtual GPU in the partition to be modified on the fly, without interfering with the others, unlike other technologies such as AMD's MxGPU [10] that opt for less costly but more inflexible static partitioning.

For Cloud Service Providers (CSPs), MIG offers a solution to resource contention in multi-tenant scenarios. However, it has traditionally been applied in a static, task-agnostic manner, meaning that the MIG partition is rarely reconfigured and does not take into account the specific features of the tasks as they are generated. MIG is currently supported by job management systems widely used in the High Performance Computing (HPC) arena, such as SLURM [11] and HTCondor [12], as well as by workload orchestrators such as Kubernetes [13]. However, their scheduling policies do not integrate strategies that combine dynamic MIG reconfiguration with job scheduling. Reconfiguring MIG partitions involves a small overhead, but it is often cost-effective as it better adapts the GPU to the demands of the pending tasks.

This gives rise to a task scheduling problem in a reconfigurable system, which requires determining both the task execution order and the partitions and GPU instances assigned to each task. This paradigm is known in the literature as *moldable scheduling*, since the scheduler determines the amount of resources to allocate to each task-i.e., *molds* them-instead of receiving these allocations as inputs, as in *rigid scheduling*. Even without reconfiguration or moldability, this scheduling

* Corresponding author.
E-mail address: jorvil01@ucm.es (J. Villarrubia).

https://doi.org/10.1016/j.future.2025.108145
Received 14 May 2025; Received in revised form 21 July 2025; Accepted 13 September 2025
Available online 16 September 2025

problem is NP-hard [14]. The addition of these two crucial dimensions to fully leverage MIG further increases the complexity, and it is a topic roughly explored in the literature. In addition, the particularities of the MIG technology for GPU partitioning impose constraints that scheduling must address, exacerbating the complexity of the problem.

In a previous work [2], we demonstrated the potential of this approach for MIG, introducing an approximation algorithm to address that scheduling problem. Our solution notably reduced the completion time of several task sets versus: not using MIG, using MIG statically (no reconfiguration), and a state-of-the-art dynamic MIG scheduler. However, the approach relied on heuristics and major simplifications given the problem's complexity, as also occurs in similar methods [15,16]. This leaves room for improvement, which we will try to exploit in this paper. The same occurs in approaches that tackle the problem with exact optimization methods, such as linear programming, which simplify the problem with key assumptions to eliminate its NP-hardness [17]. Section 2.2 outlines common simplifications and emphasizes the errors they introduce. This work goes a step further by approaching the problem without relying on overly simplistic assumptions or predefined scheduling forms.

Naturally, this involves a vast search space that cannot be exhaustively analyzed. However, for very complex problems like this, it works really well to learn by "trial and error" with a type of machine learning known as Reinforcement Learning (RL) [18]. When combined with neural networks, RL excels at tackling high-dimensional problems where the number of variables or the cardinality of the state-action space makes it computationally infeasible to represent and explore it with traditional table-based methods [19]. This fusion—known as Deep Reinforcement Learning (DRL)—has proven to be very effective for other resource management and control challenges [20,21]. Thus, it will be used in this paper for scheduling moldable tasks to optimize the use of MIG.

Beyond using a DRL approach to leverage MIG without major assumptions, this work can serve as a guide for other DRL solutions in similar environments, characterized by dynamic reconfiguration or moldability. In this regard, this paper details the steps, challenges, and solutions encountered in developing a satisfactory final model. Many of these aspects may appear in future related research and this information may be very useful. Our main contributions are summarized below:

- We formulate a moldable scheduling problem with dynamic MIG reconfiguration. We propose a Deep Reinforcement Learning framework for MIG scheduling, without major assumptions on the form of the solutions.
- We extensively evaluate our DRL scheduler, improving the makespan (completion time of all tasks) over the static use of MIG and the state-of-the-art. We also show that the obtained solutions are not too far from the optimum.
- We offer to the community a set of tools [22]: the modeling and training code of our RL agent, a scheduling tool to execute tasks on a MIG-capable GPU following the decisions of a pre-trained agent, and a graphical visualization interface for the agent's scheduling status.
- We describe the challenges and design decisions for model development, serving as actionable insights for future research on DRL-based solutions for dynamically reconfigurable or moldable environments.

The rest of the paper is organized as follows. Section 2 covers background on MIG and the related work. Section 3 presents the scheduling problem. Section 4 describes the modeling of the problem with Reinforcement Learning. Section 5 details the experimental setup. Section 6 offers an iterative refinement of the scheduler. Section 7 presents the conclusions.

2. Background

2.1. Concepts and constraints of MIG

NVIDIA incorporates the MIG technology into its latest GPU datacenter models with Ampere, Hopper, and Blackwell architectures. Specifically, the GPU models supporting MIG are A30, A100, H100, B100, and B200. A MIG-capable GPU can be partitioned into multiple virtual GPUs, called **GPU instances**. Each instance consists of one or more physically hardware-contiguous resource units called **slices** (S_i, $1 \le i \le K$, where K is the number of slices). Each slice corresponds to a GPU Processing Cluster (GPC), a hardware block that includes its own Streaming Multiprocessors (SMs), a portion of L2 cache + DRAM, and dedicated memory bandwidth. The **instance size** is determined by the number of slices it contains. Finally, the collective set of instances into which the GPU is divided is called **configuration** or **partition**.

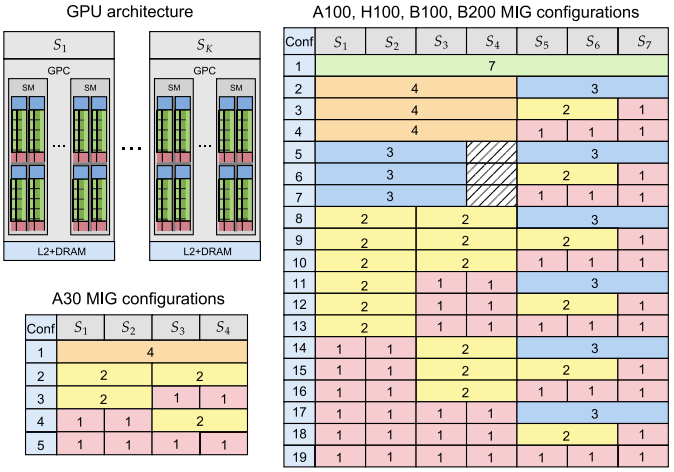

Fig. 1. Slice per instance configurations of MIG-capable GPUs.

Fig. 1 illustrates the possible configurations for current MIG-capable GPUs. As shown, not all combinations of slices form valid instances and not all combinations of instances produce feasible configurations. On the one hand, NVIDIA A30 consists of 4 slices, which can be grouped into instances of sizes 1, 2, and 4, leading to 5 possible configurations. It is clear that it is not allowed to create instances of 3 slices, nor an instance with the two central slices. On the other hand, the rest of the GPU models comprise 7 slices, which can be grouped into instances of sizes 1, 2, 3, 4 and 7, resulting in 19 possible configurations. Apart from other restrictions, in this case it is observed that configurations 5, 6 and 7 disable slice S_4 because a 3-slice instance offers the same amount of memory as a 4-slice instance. This is fine for $\{S_5, S_6, S_7\}$, since S_7 can offer twice as much memory as the other slices, but with $\{S_1, S_2, S_3\}$ it involves using the memory of S_4 while its other resources remain idle.

Configurations can be dynamically adjusted by destroying some instances and creating new ones in their place. For example, in configuration 2-2-3 (configuration number 8), instances of size 2 ($\{S_1, S_2\}$ and $\{S_3, S_4\}$), could be merged by destroying them and creating an instance with its 4 slices ($\{S_1, S_2, S_3, S_4\}$), thus transitioning to configuration 4-3 (conf. no. 2). Alternatively, the first instance of size 2 could be subdivided by destroying it and creating two new instances of size 1, thus reconfiguring to 1-1-2-3 (conf. no. 14). The instances to be destroyed must be idle, since task preemption is not allowed. However, reconfiguring an instance is independent from others, and does not affect to tasks running on them. Thus, in the previous reconfiguration examples, the instance of size 3 could be running a concurrent task. The creation or destruction of instances takes a specific amount of time, depending on their size, which must be considered to decide if reconfiguration is beneficial. Table 1 reports the measured times for these operations observed on different MIG-capable devices. These times were very consistent between runs, with deviations of no more than 2%. As such, they can be considered as constant data for scheduling on each (virtual) device, as we will do.

We also verified the effectiveness of instance isolation by measuring execution times. Experiments show negligible ($< 2\%$) timing variations

Table 1
Average time in seconds for the creation/destruction of an instance.

Instance Size	A30		A100		H100	
	Create	Destroy	Create	Destroy	Create	Destroy
1	0.11	0.10	0.16	0.20	0.16	0.21
2	0.12	0.10	0.17	0.20	0.21	0.23
3	–	–	0.20	0.21	0.33	0.25
4	0.13	0.10	0.21	0.21	0.38	0.26
7	–	–	0.24	0.22	0.42	0.26

Table 2
Task duration for each instance size (in bold, the optimal efficiency).

Instance Size	Task 1		Task 2 & Task 3	
	Time	Efficiency	Time	Efficiency
1	25	1	12	1
2	**10**	**1.25**	5	1.2
4	10	0.625	**2**	**1.5**

between single- and multi-task execution [2]. To conduct this analysis, we employed multiple GPU benchmarks from the Rodinia [23] and Altis [24] suites, which will also be used later for a comprehensive evaluation. Tests were conducted on NVIDIA A30, A100 and H100 GPUs, ensuring that task execution times remain highly predictable and are unaffected by possible co-execution patterns. Notably, prior research has presented highly accurate and efficient predictors for the execution time in MIG instances of different kind of tasks [8,9,25], achieving error margins below 2%. This capability solidifies the reliability of time-based schedulers in MIG environments, enabling these predicted times to serve as robust inputs for scheduling decisions.

2.2. Related work

MIG scheduling. Previous works have relied on heuristics [2,9,16,26,27] and optimizing methods [17], making assumptions and simplifications to reduce problem's complexity, although they induces very specific and suboptimal solutions.

A prevalent simplification involves decoupling the task molding decision (i.e., the number of resources for each task), from the final co-execution order and the specific resource allocation [2,16,17,26]. This is typically achieved by assigning to each task the instance size n that maximizes parallel efficiency $\text{eff}_n = \text{time}_1 / (n \cdot \text{time}_n)$. However, this simplification is not valid in all scenarios. For example, Table 2 presents three tasks that obtain their optimal efficiency at instance sizes 2, 4, and 4, but such an allocation is more than 40% worse for an A30 than allocating 2 slices to all tasks, as illustrated in Fig. 2. While the first allocation forces sequential execution due to resource constraints, the alternative enables parallelism, masking latency. This basic example underscores that optimizing MIG in a general case requires a joint consideration of task molding, execution order, and placement—a problem with an exponentially increasing solution space. Some works [2,26] attempt to detect and refine this problem, but start from poor schedules derived from decoupled decisions, and use fast and limited heuristics that still produce quite improvable solutions in some cases, as we will see with an example of final results in Section 6.7.

A second major simplification involves the greedy allocation of resources to pending tasks, favoring immediate assignment over strategic idleness that could enable resource merging or the exploration of larger subsets of tasks than those immediately following [2,9,16,17,26,27]. This may lead to worse results if the speedup from allocating more resources compensates for the idle time needed to merge instances, or allows for better load balancing among concurrent tasks. Also, heuristics are often quite *ad hoc*, making it difficult to generalize them to other similar systems, conditions, or optimization goals.

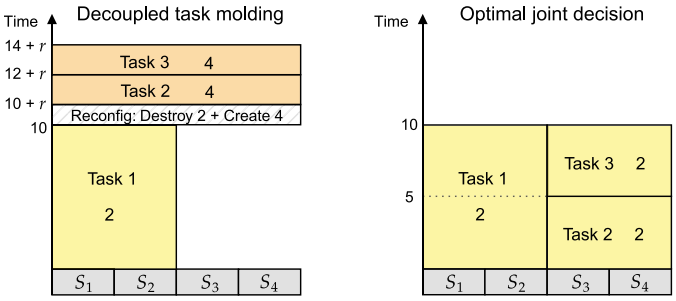

Fig. 2. Schedule of the tasks in Table 2 with independent molding according to efficiency vs. optimal joint decision.

One of our evaluation baselines will be MISO [9], which uses the MIG configuration that maximizes the sum of individual task speedups for the next queued tasks in FIFO order (reconfiguring if necessary). We will also compare against FAR [2], a 3-phase approximation algorithm that optimizes MIG scheduling of tasks in batches: phase 1 molds the tasks by creating a family of allocations where it assigns the possible instance sizes in which to execute each one; phase 2 schedules the tasks, trying to minimize the makespan for each allocation by a greedy algorithm; finally, phase 3 improve the schedule by moving or swapping some critical tasks.

As mentioned earlier, MIG is supported by job management tools such as SLURM [11] and HTCondor [12], as well as by the Kubernetes container orchestrator [13]. However, existing schedulers for those systems, most notably Volcano [28] and YuniKorn [29], do not implement a task feature-aware strategy to optimize MIG partitioning for co-execution. Their use of MIG is based on simple greedy methods that only ensure that tasks quickly satisfy their resource requirements, with worse results than the previous approaches mentioned.

The existing literature on moldable scheduling in environments different to MIG relies on similar approaches [30,31], with the limitations just discussed. To overcome this, in this paper we adopt Deep Reinforcement Learning, a technique renowned for its effectiveness in high-dimensional complex systems, and its adaptability to diverse environments and objectives. Although we will make approximations to manage the vast solution space, we will not assume a specific form of the solution, thus avoiding the issues inherent to prior methods.

Task scheduling using RL. Reinforcement Learning has been successfully applied to a wide range of task scheduling problems [32–34]. These span from classical challenges, such as Job Shop Schedule (JSS) [35,36] and Parallel Machine Scheduling (PMS) [37], to HPC clusters and a variety of heterogeneous systems [38–40]. However, tasks are typically not moldable in these works, as their resource requirements are predefined, and the system is not reconfigurable (at least, not in the way MIG is). Moreover, these approaches are not formulated as partitioning problems, as they do not have strict conditions such as contiguity of resources in MIG instances. Our proposal models these key features to leverage the full potential of MIG, discussing related approaches and challenges that can be extrapolated to similar environments.

Dynamic reconfiguration and partitioning. In the field of FPGAs, many works have explored Dynamic Partial Reconfiguration (DPR) [41], similar to MIG in terms of partitioning into isolated hardware regions with dynamic reconfiguration. DPR has been optimized with methods like those discussed for MIG [42,43], under the simplifications and limitations discussed above. Moreover, DRL has recently been successfully applied in domains with similar characteristics, such as cache partitioning [44,45] or adaptive network reconfiguration [46], which suggests suitability for our proposal.

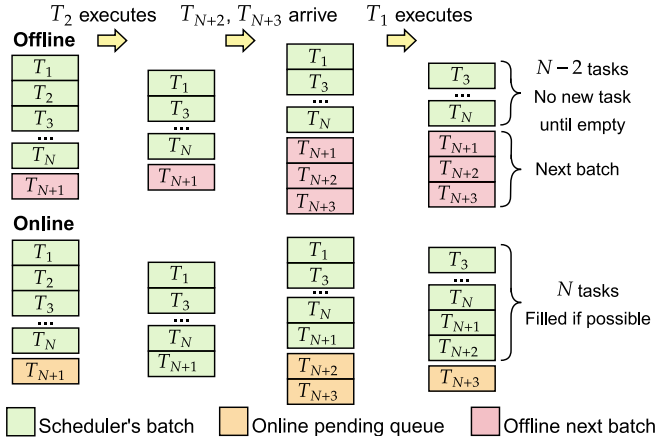

Fig. 3. Evolution of offline and online approaches with the execution or arrival of some tasks, for a batch limited to N tasks in both cases.

3. Problem Statement

In multi-device environments, incoming tasks are typically routed to one of the devices, creating queues of pending tasks on each. Although task distribution has been extensively explored [39,47,48], we focus on optimizing the scheduling of pending tasks on a MIG-capable GPU. As previously mentioned, we attempt to improve existing methods by holistically considering the dimensions of the problem. It can be approached in the following two ways, illustrated by an example in Fig. 3:

- **Offline multi-batch:** Pending tasks are organized by arrival order into batches, typically of limited size N, that the scheduler treats offline and sequentially[1]. It fully schedules each batch before processing the next one. Any task arriving during batch scheduling is part of a future batch.
- **Online:** The scheduler manages a dynamic batch of tasks, adapting its decisions to changes in the batch. Incoming tasks are included in the batch and are considered for next decisions. Schedulers that need a fixed batch size or cannot handle a large one can take as a batch the first N tasks of the pending queue (see Fig. 3). If a task moves from batch to execution, it is replaced by the next in the queue.

The offline approach is simpler and inherently prevents task starvation, regardless of the scheduling policy. In contrast, the online approach can find better co-executions between new tasks added to the batch and those that were not previously executed. However, it is more complex due to its stochastic nature. This work considers both methods, as DRL can develop robust policies for stochastic environments like the online approach, despite requiring much more training.

Input. Several papers have presented efficient and very accurate predictors for the execution times of various types of tasks at MIG instance sizes (errors up to 2%) [8,9,25]. They have used them in their schedulers without introducing significant errors. In addition, the duration of reconfiguration operations was found to be stable between runs for each instance size on a GPU (see values in Table 1). Thus, the scheduler employs a time function t_i for each task T_i, and the functions t_{create} and t_{destroy} with the duration of creating and destroying an instance (times in Table 1), depending on the instance size:[2]

$$t_i,\ t_{\text{create}},\ t_{\text{destroy}} : S_G \rightarrow \mathbb{R}^+ \cup \{\infty\},$$

where G is the GPU on which to run tasks and S_G is its set of possible MIG instance sizes for that GPU (e.g., $S_{A30} = \{1, 2, 4\}$ and $S_{A100} = \{1, 2, 3, 4, 7\}$; see Fig. 1).

Output decisions. The scheduler decides the next action based on the current batch of tasks: either reconfiguring the GPU or assigning a task to a free instance of the current configuration. In the offline approach, decisions are made only when a GPU instance is released, as no new tasks are considered in the meantime. In the online approach, the scheduler also makes decisions when a new task joins the batch if there are available resources in the GPU.

Goal. The problem can be oriented to optimize different metrics, but RL allows to easily adapt the objective by redesigning the rewards. We will minimize makespan, i.e., the time required to complete a sequence of tasks, which is the most common metric in the literature [49,50] and is linked to better joint resource utilization (our main motivation). It is especially relevant when tasks are frequently accumulated in batches, which is precisely the challenging case that presents room for improvement, rather than arriving sporadically.

4. Reinforcement Learning modeling

In this section, we present the DRL agent modeling for the MIG scheduling problem, whose high-level diagram is illustrated in Fig. 4. The environment comprises the GPU—characterized by its current MIG configuration and the tasks running on each instance ("GPU state" in the diagram)—together with the "scheduler's batch" of the task queue (introduced in the previous section). By applying time discretization and canonical encoding (which will be developed in detail in Section 4.1), those components are transformed into the state representation processed by the model: the tuple $(\text{GPU}_t, \text{Batch}_t)$ that will be specified in Section 4.1.

The model is a MLP neural network trained to approximate the agent's policy: it accepts $(\text{GPU}_t, \text{Batch}_t)$ as input and outputs a probability distribution over the actions that will be designed in Section 4.2 (red nodes of Fig. 4 whose darkness indicates the probability). As we will explain in that section, many of the actions are infeasible in some states, and the action masking technique will be used to assign a zero probability to invalid ones; this is shown on the right of Fig. 4, with all nodes turning white (probability 0), except for the only valid action, which turns into a black node (probability 1). Finally, the scheduler selects and executes an action (in Fig. 4, the *Advance* action, which will be explained in Section 4.2); and receives a reward according to the function that will be presented in Section 4.3 (-2 in this example).

4.1. State definition

To enable informed decisions, the DRL agent must jointly consider the current status of the GPU with the profile of the tasks in the batch. Formally, the state at timestep t is

$$\text{State}_t = (\text{GPU}_t, \text{Batch}_t), \quad \text{where:}$$

- The GPU state (GPU_t) captures the current MIG configuration and resource availability:

  $$\text{GPU}_t = (\text{Conf}_t, S_1^t.\text{time} \ \ldots, S_K^t.\text{time}),$$

  where Conf_t denotes[3] the current MIG configuration, $S_i^t.\text{time}$ the remaining time until slice S_i is available (0 if idle), and K the amount of GPU slices (e.g., $K = 7$ for A100, or $K = 4$ for A30).
- The batch state (Batch_t) represents the execution times of the candidate tasks for MIG instance sizes at timestep t:

  $$\text{Batch}_t = (\vec{t_1}, \vec{t_2}, \ldots, \vec{t_N}),$$

[1] While resources are busy, many tasks can be accumulated, especially with the high demand of cloud and data-centers. However, if there are not so many tasks, but free resources, a smaller batch or an online approach may be used.

[2] Often, a task cannot run on an instance due to lack of resources (usually memory). Its duration is set to infinite to avoid choosing that instance size.

[3] The superscript notation t should not be confused with exponentiation; it will indicate correspondence with timestep t when the subscript is already used.

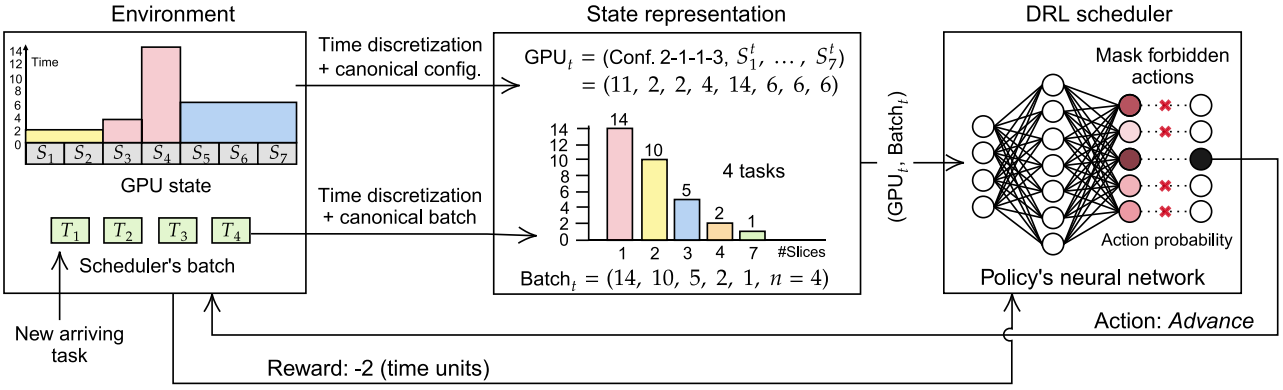

Fig. 4. High-level schematic of the DRL-based MIG scheduler.

where $\vec{t_i} = \left(t_i(1), t_i(2), \ldots, t_i\left(\max S_G\right)\right) \in \mathbb{R}^{\#S_G}$ is the execution time vector for the ith task of the batch, formed by its run-times $t_i(s)$ on each instance size $s \in S_G$, arranged in increasing order of size.

Fixed dimensionality. For compatibility with neural networks commonly used in DRL, both components (GPU_t and Batch_t) are fixed-dimensional. The slice availability vector of the GPU state has length K, fixed per GPU architecture. Moreover, the batch state is padded with zeros to an amount of N tasks: if A tasks are available, $\vec{t_i} = \vec{0}$ for $A < i \leq N$. Tasks exceeding N are deferred to subsequent batches (offline) or to a dynamic pending queue (online). A larger N provides a richer set of scheduling options, but it also enlarges the state space, which can slow down the learning process. Conversely, a smaller N simplifies the state space at the risk of excluding valuable scheduling information. N will be selected based on an empirical trade-off between learning efficiency and scheduling quality.

Data types. Another key consideration is the choice of data types for the state representation. In the case of Conf_t, an integer indicating the current configuration is the simplest and most appropriate choice. However, for the components S_i^t.time and $t_i(s)$, the decision is less obvious: should these values be continuous real numbers or may they be discretized? Although DRL can handle continuous spaces, many previous works have relied on quantized inputs to simplify the agent's learning process, particularly when representing time [16,51,52]. This introduces some approximation error, but may be necessary for our approach due to the huge number of combinations it faces. Quantization works well when high precision is not essential, as should be the case, since the times used are estimates with a slight error that in previous strategies did not practically affect the result [9,16]. In any case, both options (continuous and discretized task times) will be tested experimentally.

Canonical state representation. To facilitate agent learning, it is crucial to unify the representation of equivalent states. This makes sense when task times have been discretized, as similar states are already collapsed into equivalent states, except for reordering (in the continuous case, an exact state match is almost impossible). The batch used by the scheduler is inherently a set, so the order in which tasks appear should not affect scheduling decisions (e.g., $(\vec{t_1}, \vec{t_2})$ is equivalent to $(\vec{t_2}, \vec{t_1})$). However, the neural network approximating the policy receives the input as a vector, where task ordering matters (hence, the use of tuple notation for Batch_t). For times quantized to M possible values, each task is represented as a vector $\vec{t_i} = \left(t_i(1), \ldots, t_i\left(\max S_G\right)\right) \in \mathbb{N}_M^{\#S_G}$, interpreted in base M to obtain an associated natural value $\text{Num}\left(\vec{t_i}\right)$. These numbers allow sorting the tasks to obtain a canonical representation of the batch that simplifies the agent's learning:

$$\text{Batch}_t = \left(\vec{t_{i_1}}, \ldots, \vec{t_{i_N}}\right), \quad \text{with:}$$

$$\{i_1, \ldots, i_N\} = \{1, \ldots, N\}, \quad \text{Num}\left(\vec{t_{i_j}}\right) \geq \text{Num}\left(\vec{t_{i_{j+1}}}\right), \ \forall\, 1 \leq j < N.$$

However, this representation is redundant and can be simplified by including the number of available tasks of each class, rather than repeatedly listing their vectors. A *task class* is each of the discretized time vectors $\vec{t_{i_j}}$, so that two tasks with the same associated vector are of the same class. The total number of tasks represented remains capped to N (i.e., the sum of the task counts does not exceed N), and prior aspects such as sorting or padding with zeros still apply:

$$\text{Batch}_t = \left(\left(\vec{u_{i_1}}, n_{i_1}^t\right), \ldots, \left(\vec{u_{i_N}}, n_{i_N}^t\right)\right), \quad \text{with:}$$

$$U = \{\vec{u_{i_1}}, \ldots, \vec{u_{i_N}}\} = \{\vec{t_1}, \ldots, \vec{t_N}\}, \qquad \text{(Same unique tasks)}$$

$$n_{i_j}^t = \#\{i \ : \ \vec{t_i} = \vec{u_{i_j}}\}, \qquad \text{(Counts of tasks)}$$

$$\text{Num}\left(\vec{u_{i_j}}\right) > \text{Num}\left(\vec{u_{i_{j+1}}}\right), \quad 1 \leq j \leq \#U, \qquad \text{(Unique and sort)}$$

$$\vec{u_{i_k}} = \vec{0}, \ n_{i_k}^t = 0, \quad \#U < k \leq N. \qquad \text{(Padding)}$$

Useless configurations. As explained in Section 2.1, current MIG-capable GPUs (except NVIDIA A30) offer three configurations with the instance $\{S_1, S_2, S_3\}$ (labels 5, 6, and 7 in Fig. 1), where slice S_4 is disabled by using its memory but not its computational resources. Except for marginal energy savings, these configurations are worse in every way than those that fully utilize S_4 via $\{S_1, S_2, S_3, S_4\}$. Our tests indicate that an instance of size 4 runs at least as fast as an instance of size 3, making size 4 preferable when size 3 disables four slices. Fig. 7 illustrates the speedup with respect to instance size for several benchmarks of the Rodinia and Altis suites, and in particular shows higher or equal speedup with 4 slices than with 3. This evidence supports eliminating the mentioned configurations (5, 6 and 7 of Fig. 1), thus reducing the solution space.

Equivalent configurations. Different configurations can be equivalent for scheduling, as they represent identical scenarios under instance reordering. For example, configurations 2-1-1-3 and 1-1-2-3 (labels 11 and 14 in Fig. 1) become the same when the instance order is changed from (I_1, I_2, I_3, I_4) to (I_2, I_3, I_1, I_4); that is, the slice basis is transformed to $(S_3, S_4, S_1, S_2, S_5, S_6, S_7)$. Moreover, this base-change does not affect the scheduling possibilities, since every possible GPU configuration has a preimage in the original base (there is a bijection between their solution spaces). However, this property does not hold for all instance reorderings. For example, consider configurations 2-2-1-1-1 and 1-1-2-2-1 (labels 10 and 15), which become the same by reordering its instances as $(I_3, I_4, I_1, I_2, I_5)$, corresponding to a new base $(S_3, S_4, S_5, S_6, S_1, S_2, S_7)$. In one configuration, merging instances of size 2 yields 4-1-1-1, while in the other, this merging is not permitted. This discrepancy is illustrated by reversing the change of base on the 4-1-1-1 configuration, which results in 1-1-4-1, that violates the MIG constraints (see Fig. 1). Accordingly, the following formal definition is established:

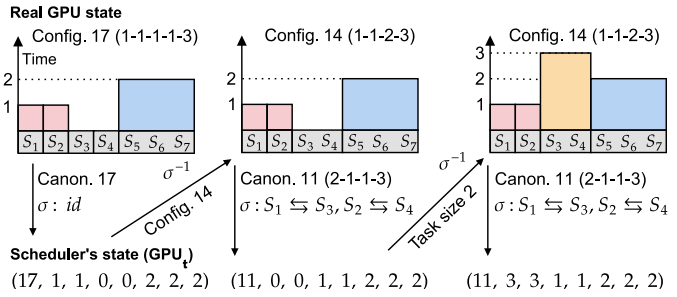

Fig. 5. Conversion between the real GPU state and the scheduler's canonical state by changing basis with σ and σ^{-1}.

Definition 1. Let two configurations $C = (c_1, \dots, c_K)$ and $C' = (c'_1, \dots, c'_K)$, where c_i is the size of the instance to which the ith slice of C belongs[4]. C and C' are **equivalent** if and only if:

1. Equals under reordering: There exists a permutation σ over $\{1, \dots, K\}$ such that $c_{\sigma(i)} = c'_i,\ \forall\, 1 \le i \le K$.
2. Same configurations: Each MIG valid configuration $C'' = (c''_1, \dots, c''_K)$ has a preimage via σ. That is, $\left(c''_{\sigma^{-1}(1)}, \dots, c''_{\sigma^{-1}(K)}\right)$ is also a valid MIG configuration[5].

Equivalent configurations represent the same scheduling scenario in different bases, but each adds extra dimensionality to the state space. Thus, choosing a canonical representative for each equivalence class is beneficial; it inherently tells the scheduler that states differing only by order are equivalent, simplifying its learning. We select the canonical configuration as the one with the highest lexicographical value; for example, partition 2-1-1-3 is greater than 1-1-2-3 since its first element (2) exceeds that of the other (1). This canonical form is used by the scheduler even if the GPU is actually partitioned in another equivalent way. For example, if the GPU is configured as 1-1-1-1-3, and the third and fourth instances just finish, the device can be immediately reconfigured to 1-1-2-3, but the state handled by the scheduler will be in the canonical form 2-1-1-3. This is shown in Fig. 5, performing the conversion from the real state to the canonical one with the permutation σ between them:

$$\text{GPU}_t = \left(\text{Canon}(\text{Conf}_t), S^t_{\sigma(1)}.\text{time}, \dots, S^t_{\sigma(K)}.\text{time}\right).$$

The scheduler makes decisions in the canonical configuration, and its decisions are translated back to the real configuration with the inverse permutation σ^{-1}. This is illustrated in Fig. 5 by assigning a new task to the size 2 instance (first and second slices for the scheduler), which is translated by σ^{-1} to the real instance of size 2 on the GPU (third and fourth real slices). For the NVIDIA A30, this method reduces one of the five possible partitions since the configurations 2-1-1 and 1-1-2 (labels 3 and 4 in Fig. 1) are equivalent. For other GPUs (A100, H100, B100, and B200), the number of partitions is reduced from sixteen to thirteen since the following pairs are equivalent: 2-1-1-3 vs. 1-1-2-3 (labels 11 and 14), 2-1-1-2-1 vs. 1-1-2-2-1 (labels 12 and 15), and 2-1-1-1-1-1 vs. 1-1-2-1-1-1 (labels 13 and 16).

Space cardinality. The real number of states that the problem may pose is difficult to estimate, since it depends heavily on the workload type. In the continuous representation, it is potentially infinite, and with discretization it is huge but many representable states never occur. For example, Fig. 7 shows that instance-size relative speedups do not decay as slices increase, ensuring that non-decreasing time workloads do not happen (i.e., $t_i(s) \ge t_i(s')$ for $s < s' \in S_G$). Although calculating the state space after excluding non-decreasing tasks would still overestimate reality, it illustrates the enormous potential combinations — a key motivation for earlier optimizations. Omitting tedious details, the state space's cardinality for NVIDIA A100 is given by $\sum_{i=1}^{N} C\big(C(M+4,5), i\big) \cdot 7M^6$, where $C(n,k)$ is the combinatorial notation, N is the number of tasks in the batch, and M is the number of discretization levels. With just $N = 7$ and $M = 7$ there are $\sim 10^{17}$ possibilities. In practice, only a small fraction of these states occur, and even fewer are frequent enough to be truly relevant. Moreover, modern learning algorithms do not require exhaustive exploration of the state space to find patterns that offer robust performance.

Clearly, the complexity of the state space grows with the number of configurations. However, as discussed, there is neither the need nor the opportunity to exhaustively explore a vast number of states to discover effective policies when using modern guided methods such as DRL. Furthermore, in the short to medium term it is unlikely that GPUs will support many more configurations, since the current partitioning already provides more than enough flexibility for co-executing various applications. In fact, the last three NVIDIA generations (Ampere, Hopper, and Blackwell) have maintained the same MIG scheme, as shows the long list of GPU models in Fig. 1. So far, GPUs has been scaled by increasing the resources per slice (more SMs and memory), instead of adding more slices. Also, our proposal could be realized in any subset of configurations.

4.2. Action definition

Possible actions consist of reconfiguring the GPU or executing batch tasks on an instance of the current configuration (in the online approach, the next pending task is added to the batch if available). Actions can only be performed when at least one instance in the current configuration is free (that is, $\exists\ 1 \le i \le K \mid S^t_i.time = 0$): reconfiguration or the execution of a new task is impossible when all instances are occupied. This constraint defines the decision timesteps for the RL agent. To model these timesteps, we include an additional action *Advance*, which moves the GPU time forward until the next instance becomes free. Notably, the *Advance* action may be taken consecutively or even when some instances are available, allowing the model to maintain idle slices until enough tasks accumulate to merge them into a larger instance (one of the motivating aspects over previous proposals in Section 2.2). Thus, for each state, the model performs one of these actions, modifying the state as detailed formally below:

Advance

Precondition: It is available if the GPU is not completely idle, meaning that not all slices have a remaining time of zero (i.e., $S^t_i.\text{time} \neq 0$ for some $1 \le i \le K$).

Result: This action advances the decision time until the next instance completes:

$$S^{t+1}_i.\text{time} := S^t_i.\text{time} - \text{next\_time}, \quad \forall\, 1 \le i \le K,$$

where $\text{next\_time} = \min\left\{S^t_i.\text{time} : S^t_i.\text{time} > 0,\ \ 1 \le i \le K\right\}$. This action is equivalent to wait for the shortest remaining task completion event.

Reconfiguration to C

Precondition: The slices to be modified from the current scheduler configuration $\text{Canon}(\text{Conf}_t) = (c^t_1, \dots, c^t_K)$ to the target configuration $C = (c_1, \dots, c_K)$ must be free (i.e., their remaining times must be 0):

$$S^t_i.time = 0, \quad \forall\, 1 \le i \le K \text{ such that } c^t_i \neq c_i.$$

[4] For clarity, the configurations are represented in the text with a compact notation, listing the size of each instance in order (e.g., 4-1-1-1). However, for this formal definition, it is preferable to list the instance size of each slice (e.g., $(c_1, c_2, c_3, c_4, c_5, c_6, c_7) = (4, 4, 4, 4, 1, 1, 1)$).

[5] The injectivity is trivially inherited from the injectivity of σ. Thus, only suprayectivity is required for MIG configurations in point 2.

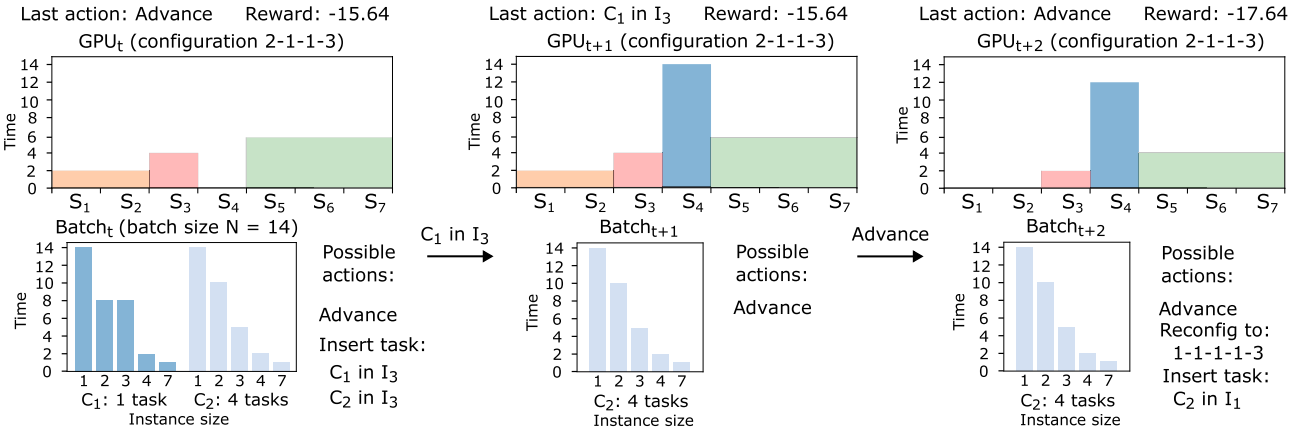

Fig. 6. Scheduler's state and its evolution after 2 actions. It graphically illustrates the GPU (GPU_t), the batch ($Batch_t$) and a list of possible actions.

For example,[6] to change 4-3 (4-4-4-4-3-3-3) to 4-2-1 (4-4-4-4-2-2-1), we ensure the last three slices are free to remove the size 3 instance and create size 2 and 1 instances. Two or more consecutive reconfiguration actions do not make sense, since the model could transition directly to the last configuration at a lower cost.

Result: Instances that are in Canon($Conf_t$) and not in C are destroyed, and instances that are in C and not in Canon($Conf_t$) are created (applying σ_t^{-1} to go from the representations to the real GPU instances). The remaining time for reconfigured slices is increased by the time required to destroy the previous instance and create the new one:

$$S_i^{t+1}.\text{time} \mathrel{+}= t_{\text{destroy}}(c_i^t) \quad \text{if } c_i^t \neq c_i, \quad 1 \leq i \leq K,$$
$$S_i^{t+1}.\text{time} \mathrel{+}= t_{\text{create}}(c_i) \quad \text{if } c_i^t \neq c_i, \quad 1 \leq i \leq K.$$

The next configuration is set to C, i.e., $Conf_{t+1} := C$, computing the new canonical configuration Canon($Conf_{t+1}$) and updating the permutation σ_{t+1}.

Execute task $\vec{t}_{i_j}$ on instance I_m
Precondition: The task does not belong to the padding, i.e., $1 \leq j \leq N$ and $\vec{t}_{i_j} \neq \vec{0}$. Moreover, the instance I_m must be one of the available instances in the current configuration; specifically, the canonical configuration Canon($Conf_t$) must contain at least m instances such that each slice $S \in I_m$ is free: $S.\text{time} = 0 \; \forall S \in I_m$.

Result: The counter n_{i_j} is decremented, removing the task from the batch if it reaches zero (last task with those times):

$$n_{i_j}^{t+1} := n_{i_j}^t - 1, \quad \text{removing } (\vec{u}_{i_j}, n_{i_j}^{t+1}) \text{ if } n_{i_j}^{t+1} = 0.$$

If the task u_{i_j} has been removed, the canonical form of $Batch_{t+1}$ is recalculated. Task T_{i_j} begins execution on the GPU instance I_m (using σ_t^{-1} to identify the real instance on the GPU). The state is updated by adding the task's duration on that instance size to the remaining time of its slices:

$$S_i^{t+1}.\text{time} \mathrel{+}= t_{i_j}(\#I_l), \quad \forall S_i \in I_l.$$

Each action can be mapped to a different integer that uniquely identifies the action to be executed. In particular, the number 0 is the *Advance* action; the following consecutive integers correspond to each of the possible reconfigurations; and the subsequent integers are assigned to the execution of each task in every possible instance. Given that there are up to N tasks in the batch (maybe there is no padding) and up to K instances in the current configuration (configuration with all instances of size 1), there are up to $N \cdot K$ possible task execution actions. Thus, the total number of possible actions is:

$$\#\text{Actions} = 1 + \#\text{Configurations} + N \cdot K.$$

For current GPUs, this gives $5 + 4N$ possible actions on NVIDIA A30 and $14 + 7N$ actions on all other GPU models (with configuration reductions made). However, it should be noted that many actions cannot be executed in every state due to the preconditions mentioned above.

Fig. 6 presents an example of the scheduler state, and its evolution with two actions, as visualized by the graphical tool provided [22]. Although the initial batch contains $N = 14$ tasks, many have already been processed using the offline approach until timestep t, leaving only 5 tasks of two classes (1 task of class C_1 and 4 tasks of class C_2). At timestep t, only 3 of the $14 + 7 \cdot 14 = 112$ total actions are available: *Advance* and insert each class of task into the free instance. Many actions are forbidden because the batch is not full, there are up to 4 repeated tasks of class C_2, and no reconfigurations are available with the entire GPU used except for one instance. Similarly, only *Advance* is possible at timestep $t + 1$, after a task is placed on the free instance. Finally, at timestep $t + 2$, the first instance is freed, and 2 more possible actions arise: execute the class of task left on it, or reconfigure the GPU by splitting that instance (2-1-1-3 to 1-1-1-1-3). In summary, the figure illustrates that normally very few of the represented actions will be feasible.

An option to limit the action space in each state is to heavily penalize infeasible actions so that the RL agent learns to avoid them. However, given the vast state space, this approach may require extensive training without guaranteeing that an impossible action will never be chosen. Although a decision could be made using only the probabilities of feasible actions, incorporating the probabilities of infeasible actions into model training distorts the process. To address this issue, we make the model aware of feasible actions in each state by employing the action mask technique [53,54]. As shown in prior work [53], this technique outperformed penalty-based methods by achieving early convergence—within the first 10% of timesteps in some cases—while penalties failed to prevent prohibited actions even after full training. Our tests revealed the same: after hundreds of millions of timesteps providing stable convergence on action-masked results (Section 6), penalties still allowed forbidden actions, resulting in extremely poor average rewards and learning issues. Some RL training algorithms such as PPO are already implemented with these masks in popular frameworks such as Stable-Baselines3[7], and others can be easily adapted using the guidelines provided in the literature.

[6] For clarity, indices from 1 to K are in the scheduler's reference system. This means that S_1 may not be the actual GPU first slice but rather to $S_{\sigma_t^{-1}(1)}$.

[7] https://sb3-contrib.readthedocs.io/en/master/modules/ppo_mask.html

4.3. Reward design

Although, in theory, the number of timesteps can be infinite, in practice, training is conducted in successive epochs with a finite number of steps. The most popular frameworks model this process as episodes, where a stopping condition eventually signals the end. The objective of the RL model is to learn a policy π that maximizes the cumulative time-discounted reward over the episode: $\max_\pi \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t r_t \mid \pi\right]$, where $\gamma \in (0, 1]$ is the discount factor, r_t is the reward for the action of step t, and T is the episode duration. Setting $\gamma = 1$ and defining the reward of each action as the time spent with it, the final reward corresponds to the completion time of the last task (makespan) if the episode concludes after placing all tasks and spending time until there are no tasks left executing. Thus, the stopping condition of an episode is given by: $S_i^T.time = 0$ for all $1 \le i \le K$ and $\text{Batch}_T = (\vec{0}, \dots, \vec{0})$. The only action that allows time to progress is *Advance*, which is rewarded with a negative time value so that the maximization objective ultimately becomes makespan minimization:

$$r_t = \begin{cases} -\text{next\_time}_t & \text{if } a_t = Advance, \\ 0 & \text{otherwise,} \end{cases}$$

with $\text{next\_time}_t = \min\left\{S_i^t.\text{time} : S_i^t.\text{time} > 0,\ 1 \le i \le K\right\}$. Note that the *Advance* action is the only available option when all GPU slices are occupied or were reconfigured in the previous step. This ensures that episodes cannot be infinite or absurdly long, a situation that could arise from consecutive reconfigurations of reward 0, but such actions are explicitly disallowed.

5. Experimental setup

5.1. Workload datasets

To thoroughly evaluate the proposal, we use real workloads from the Rodinia [23] and Altis [24] benchmark suites, and synthetic task times generated on demand. In our case, synthetic times are essential for training because one of the key aspects is to have a large amount of different input data to train the models. In addition, they allow us to explore particularly illustrative or extreme scenarios that help to refine it, while facilitating a more complete and varied validation in a wider range of situations. Although RL modeling is general and serves for all current and future MIG capable GPUs, all evaluation experiments will be carried out on a NVIDIA A100, whose partitioning scheme is the most complex and challenging at present (more than NVIDIA A30, and equivalent to more modern GPUs).

5.1.1. Real benchmark tasks

We aim to include applications with diverse computational and memory access patterns. Generally, we have used benchmarks with their default input, but in some cases the input size has been adjusted (either increasing or decreasing their input sizes) to create more diverse workloads. Overall, we have utilized 18 real tasks based on state-of-the-art benchmarks. Fig. 7 shows the speedups observed on NVIDIA A100 as a function of the MIG instance size (relative to a single slice) for 14 selected benchmarks[8]. Some of these tasks scale marginally from 2 slices (e.g., *Backprop*), others exhibit scalability up to a certain instance size before reaching a performance plateau (e.g., *Gaussian* up to 4 slices), and others scale efficiently across all GPU slices (e.g., *SRAD*). There are also tasks that superscale, since they are memory bound, and increasing the available memory with more slices improves their performance above linear (e.g. *Raytracing*). The duration of the tasks is also varied, with some like *Myocyte* lasting less than a second, while others like *Raytracing* take minutes.

[8] The 4 benchmarks not shown exhibit considerable overlap with some of the displayed ones, and their inclusion would not add further information.

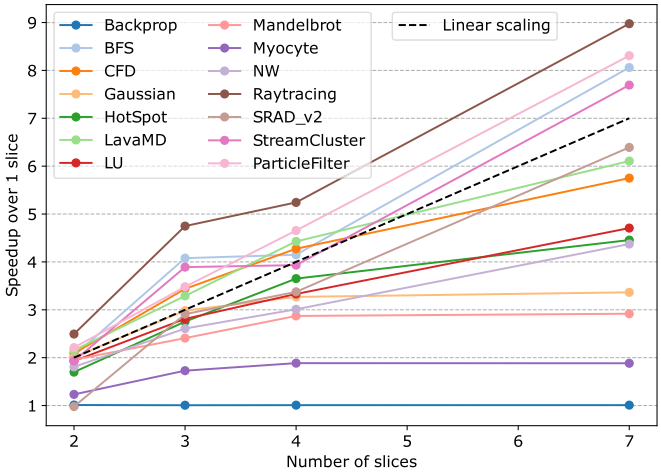

Fig. 7. MIG speedup of selected Rodinia and Altis kernels on NVIDIA A100.

These workloads will be used for testing, as a tool to validate the real applicability of the model, without using them in training. The automatic generation of synthetic workloads for training will be inspired by the behaviors exhibited by benchmarks (poor scaling, scaling up to a certain instance size or superscaling), but with some randomness to avoid sticking exclusively to that particular data, guaranteeing a richer exploration. From the 18 available benchmarks, we generate 100 different random test subsets containing at least N tasks and another 100 random different test subsets with less than N tasks. This allows us to evaluate not only the joint scheduling of all tasks but also different groupings, including scenarios where the number of tasks is insufficient to fill a batch. Additionally, for each subset, we consider five different permutations to assess the impact of task arrival order in both offline and online scheduling approaches.

We provide a MIG scheduling tool [22] that determines the sequence of actions to execute on a set of tasks by querying a pre-trained RL agent, and then performs those actions on the GPU. To achieve this, the tool waits for the required slices to be idle before executing any action, using synchronization mechanisms; task execution or reconfiguration reserves the involved slices exclusively and releases them upon completion, thereby blocking subsequent actions that require the same slices. The process is easily and faithfully simulated due to the predictability of the times resulting from the isolation of the instances, and the inherent determinism of MIG. In fact, the agent performs internal simulations during training based on the state transitions described in Section 4.2, where the progression of time with the *Advance* action corresponds to the next resource release event. It is important to check that the simulation is faithful to reality as expected, so Table 3 compares the end times of a set of benchmarks co-executed with MIG on an NVIDIA A100 against the simulated results, along with the corresponding percentage deviations. As illustrated, the deviations are minimal, which supports the simulation's accuracy. We will use the simulation to evaluate the proposal over a large number of workloads, in particular over synthetic times, which will facilitate model debugging and refinement.

5.1.2. Task time generator

The synthetic generator was introduced in our previous work [2]. However, to ensure this paper is self-contained and easier to follow, we revisit its fundamental concepts. Synthetic times aim to replicate the behavior observed in real workloads (Fig. 7), where task performance scales efficiently (linearly or superlinearly) up to a certain instance size, beyond which improvements become negligible. In some cases, this critical instance size corresponds to the full GPU, so the task scales effectively across all resources. The generator takes as inputs:

- The number of tasks n to generate.

Table 3
Comparison of simulated vs. real end execution times, in a batch of 9 Rodinia kernels scheduled by the RL agent on NVIDIA A100.

Kernel	End time (s)		Diff. (%)
	Sim.	Real	
PathFinder	3.45	3.53	2.27
LavaMD	7.34	7.29	−0.69
Huffman	7.81	7.89	1.01
NW	8.63	8.71	0.92
PathFinder	8.64	8.65	0.12
LU	8.66	8.65	−0.12
HotSpot	8.73	8.70	−0.34
HeartWall	9.07	9.11	0.44
Gaussian	11.01	11.22	1.87

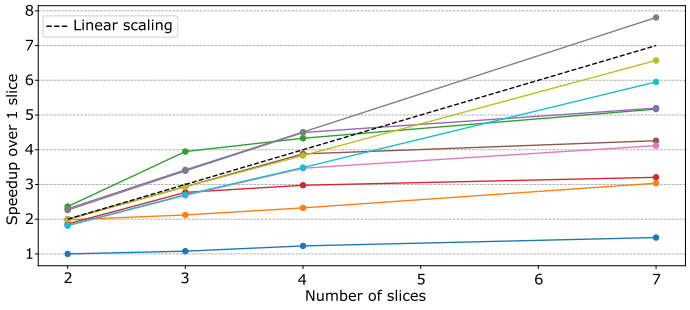

Fig. 8. Synthetic speedups for $n=10$ tasks in NVIDIA A100 MIG scheme, using $p_1=p_2=10\%$, $p_3=20\%$, $p_4=p_7=30\%$ and $p_{\text{sup}}=50\%$.

- The percentage p_s of tasks that scale well up for each instance size $s \in S_G$: near-linear speedup for $s' \leq s$ and sub-linear speedup for $s' > s$.
- The percentage p_{sup} of tasks that initially exhibit super-linear speedup when transitioning from 1 to 2 slices, as commonly observed in memory-bound workloads.

Furthermore, at each transition to a larger instance size, there is a 0.3 probability that the task ceases to be memory-bound, thus no longer exhibiting superscaling behavior. For each task, a duration for an instance with one slice, denoted as $t_i(1)$, is generated using a uniform distribution $U(t_{\min}, t_{\max})$, where $t_{\min}$ and $t_{\max}$ are configurable parameters. Then, the time for an additional slice, $t_i(s+1)$, is generated based on the previous value $t_i(s)$ according to the type of speedup between them:

$$t_i(s+1) = \frac{s+r}{s+1} \cdot t_i(s), \quad \text{with } r \text{ taken random as}$$

$$r \in \begin{cases} N(-0.25, 0.25)_{[-0.5,0]} & \text{if super-linear speedup,} \\ N(0.1, 0.1)_{[0,0.2]} & \text{if near-linear speedup,} \\ N(0.75, 0.25)_{[0.5,1]} & \text{if sub-linear speedup,} \end{cases}$$

where $N(\mu, \sigma)_{[a,b]}$ is a normal distribution of mean μ and standard deviation σ clipped to $[a, b]$. For $r=0$, the speedup is linear, while higher r values reduce improvement. It generates $r<0$ for super-linear, $r\approx 0$ for near-linear, and $r>0$ for sub-linear speedup. Values are clipped to ensure the desired speedup type (e.g., no $r>0$ for super-linear cases).

This synthetic generation strategy of tasks allows replicating behaviors similar to those observed in benchmarks with some variability, while offering the flexibility to produce specific workloads if needed. Fig. 8 presents the speedups for a set of synthetically generated execution times, closely resembling the patterns observed in the benchmarks shown in Fig. 7.

5.1.3. Synthetic workloads

Training is performed with task times obtained successively with the generator, according to the workload specification among those listed below. For evaluation, the generator is used to create datasets covering varied and boundary cases. Specifically, 1000 datasets were created for each of the below workload types, with $n = 100$ tasks per dataset:

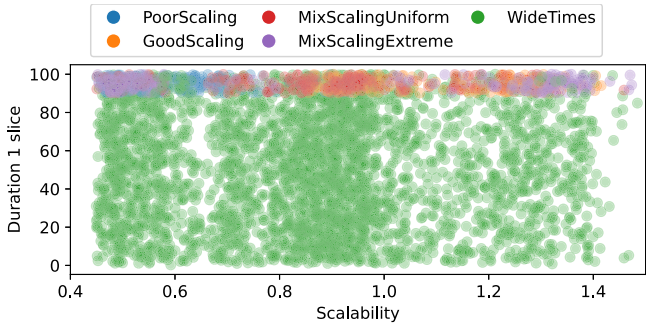

Fig. 9. Synthetic tasks of different workloads according to their scalability and duration with 1 slice.

- PoorScaling: Scale up to few slices (at most 2 slices). It is defined by $(p_1, p_2, p_3, p_4, p_7)=(50, 50, 0, 0, 0)\%$, $p_{\text{sup}}=25\%$ and $(t_{\min}, t_{\max})=(90, 100)$ seconds.
- GoodScaling: Scale to nearly all GPU slices. It is defined by $(p_1, p_2, p_3, p_4, p_7)=(0, 0, 0, 50, 50)\%$, $p_{\text{sup}}=75\%$ and $(t_{\min}, t_{\max})=(90, 100)$ seconds.
- MixScalingUniform: Varying scaling capabilities. It is defined by $(p_1, p_2, p_3, p_4, p_7)=(20, 20, 20, 20, 20)\%$, $p_{\text{sup}}=50\%$ and $[t_{\min}, t_{\max}]=[90, 100]$ seconds.
- MixScalingExtreme: Varying scaling capabilities, polarized to scale very poorly or very well. It is defined by $(p_1, p_2, p_3, p_4, p_7)=(45, 5, 0, 5, 45)\%$, $p_{\text{sup}}=50\%$ and $(t_{\min}, t_{\max})=(90, 100)$ seconds.
- WideTimes: Times span a wide range $[t_{\min}, t_{\max}]$, with very short tasks scheduled alongside much longer ones. It is combined with the most varied scalability configuration. It is defined by $(p_1, p_2, p_3, p_4, p_7)=(20, 20, 20, 20, 20)\%$, $p_{\text{sup}}=50\%$ and $(t_{\min}, t_{\max})=(1, 100)$ seconds.

Fig. 9 illustrates the distribution of 2 of the 1000 datasets generated for each workload, each containing 100 tasks. For the WideTimes, however, 30 of the 1000 datasets are shown, as these cover a broader range of options. Each task is positioned according to its scalability level, which is quantified as the average parallelization efficiency across different instance sizes. Formally, for a given task T_i, the scalability is defined as:

$$\text{scalability}(T_i) = \frac{1}{\#S_G - 1} \sum_{s \in S_G \setminus \{1\}} \frac{t_i(1)}{s \cdot t_i(s)}.$$

Furthermore, the intrinsic nature of each task is characterized by its duration. To capture this in a single dimension, we consider the execution time for one slice, $t_i(1)$, since the durations corresponding to larger instance sizes are inherently reflected in the scalability measure. It can be observed a varied distribution of loads that effectively covers a wide spectrum of scalability levels: from 0.4 (i.e., just 40% average efficiency) to 1.4 (i.e., 140% average efficiency). The WideTimes workloads span a range of durations, while other specific loads are tailored for different scaling scenarios (some of which are deliberately exaggerated) for debugging purposes in the scalability dimension, so they are generated within a narrower time range.

5.2. Evaluation metrics

To evaluate the model, we compare it to other state-of-the-art approaches using a metric that implicitly returns a maximum distance to the optimal makespan. For that, we use a lower bound on the makespan for a set of n tasks, computing a lower bound on the total area occupied

in a 2D representation such as Fig. 2 (horizontal axis is slices and vertical is time), and evenly partitioning it among the K slices on the GPU. Since the total area is at least the sum of the area occupied by tasks (A_{task}) and the area occupied by reconfigurations ($A_{\text{reconfig.}}$), we lower bound each term based on the subset S of instance sizes used by tasks, minimizing for all possible subsets:

$$l\_bound = \frac{\min_{S \neq \emptyset \subseteq S_G} \left(L_{\text{tasks}}(S) + L_{\text{reconfig.}}(S) \right)}{K} \leq \frac{A_{\text{tasks}} + A_{\text{reconfig.}}}{K} \leq \frac{A_{\text{total}}}{K} \leq \text{opt. makespan},$$

where S is the subset of instance sizes used by the tasks in the batch. To calculate the task area bound, we add a lower bound on the area of each task, minimizing for the possible instance sizes $s \in S$ the sum between the area occupied by it ($s \cdot t_i(s)$) and the area that necessarily remains idle ($\text{Idle}(s, S) \cdot t_i(s)$):

$$L_{\text{tasks}}(S) = \sum_{i=1}^{n} \min_{s \in S} \left(s \cdot t_i(s) + \text{Idle}(s, S) \cdot t_i(s) \right).$$

$\text{Idle}(s, S)$ calculates a number of slices that will necessarily be idle while a task executes on an instance of size s, where S is the subset of sizes used by all tasks. To do this, for each possible configuration C that includes an instance of size s, the instance sizes s' that are necessarily unoccupied since they are not in S are added together, making the minimum among all possible configurations:

$$\text{Idle}(s, S) = \min_{s \in C \text{ of MIG}} \sum_{s' \in C,\ s' \notin S} s'.$$

Finally, to compute a lower bound of the reconfiguration, $L_{\text{reconfig.}}(S)$, we simply sum the area associated with the creation and destruction of each instance size used $s \in S$ once:

$$L_{\text{reconfig.}}(S) = \sum_{s \in S} s \cdot \left(t_{\text{create}}(s) + t_{\text{destroy}}(s) \right).$$

We use this lower bound to calculate the ratio r_{opt} of the makespan produced by the model, which serves as an overestimated bound of the deviation from the optimum:

$$r_{\text{opt}} = \frac{\text{scheduler's makespan}}{l\_bound}.$$

Although the times may be discretized for the model observations, obviously the makespan for the above metric is performed with the original continuous times, so that the error derived from the discretization will be included in the ratio. The closer the ratio is to 1 the closer the scheduled makespan is to the optimum; for example, $r_{\text{opt}} = 1.2$ means a difference of at most 20% relative to the optimal solution (possibly less). For clarity, we will report this percentage p_{opt}, calculated as follows:

$$p_{\text{opt}} = (r_{\text{opt}} - 1) \cdot 100$$

For the offline approach, the scheduler does not process other tasks from the dataset until all tasks in the batch have been planned, so we consider each batch as a set of tasks to compute the metric. In this case, the dataset's metric is obtained by averaging the metrics computed of its batches. In the online approach, after N tasks scheduled (the batch size), the model has handled unscheduled tasks due to replacement (see Fig. 3). Computing the metric only with scheduled tasks, or with all handled tasks, would be unfair. The most equitable would be to compare against the subset of N handled tasks with the smallest lower bound, but this is computationally prohibitive due to the exponential number of possibilities. Instead, we will directly compare the makespan achieved by the online RL method with that of other approach A on the same task set, reporting the results as percentage improvements:

$$p^{A}_{\text{online}} = \left(\frac{\text{makespan of } A}{\text{makespan of online RL agent}} - 1 \right) \cdot 100$$

5.3. Comparison baselines

The model will be compared against the following proposals:

- FAR: A heuristic-based approximation MIG scheduler presented in our previous work [2]. It works offline, so it will serve to compare the RL agent exclusively in that approach. According to our observations, its results are better than: previous state-of-the-art proposals, execution without using MIG, and MIG without taking advantage of dynamic reconfiguration.
- MISO: A state-of-the-art MIG scheduler [9] that prioritizes queued tasks in FIFO order, running each with the next possible configuration that maximizes the cumulative speedup of running instances.
- FixBest: MIG without reconfiguration, meaning that a single partition is used for the entire dataset. Tasks are allocated to instances in FIFO order. An ideal scenario is assumed in which the chosen configuration is optimal for the dataset, i.e. the makespan is computed for all configurations, and the one with the lowest makespan is selected.
- NoMIG: Tasks are allocated to the entire GPU without MIG partitioning and task co-execution.

6. Evaluation and refinement

This section presents the evaluation of our proposal, where numerous hyperparameters require fine-tuning. To demonstrate the impact of some hyperparameters, we include plots that depict the evolution of the p_{opt} metric during training. Additionally, we refer to the results summarized in Table 4, which shows the p_{opt} values after 200 million timesteps for various models and workloads. In this table, the rows correspond to the different workload types, while the columns compare the baselines with the various refined versions explained in this section. Since the p_{opt} metric is better suited to an offline approach, we adopt this method for our experiments until Section 6.6, where we will perform an online evaluation and compare its results with those of the offline approach.

6.1. Model training strategy

For training the RL agent in the environment described in Section 4, we evaluated various frameworks, algorithms, and neural network architectures. Naturally, the choice depends on preliminary tests before conducting longer training sessions with the selected setup. The set of training options had to be constrained due to the high cost of long training runs, especially with certain modeling decisions still open for experimental validation, which act as hyperparameters (N, M, offline vs online approach, etc.). In particular, we adopt the offline approach for this study since the p_{opt} metric is more accurate in that case, as explained above, and there is no reason to suspect that the approach is decisive in these validations.

Regarding frameworks, we tested StableBaselines3 [55] and RLlib [56], finding better performance with StableBaselines3. This is probably because RLlib is designed for distributed learning in clusters and typically requires multi-agent environments with quite large network architectures to be fully leveraged.

We evaluated two widely adopted DRL training algorithms, representing distinct paradigms: Deep Q-Learning (DQN) [57], which estimates state-action values, and Proximal Policy Optimization PPO [58], which directly optimizes the policy by gradient descent, refining it through an actor-critical method with the value estimates. As Fig. 10 shows, PPO surpasses DQN in convergence speed and final performance for GoodScaling workload, batch size $N = 14$, discretization levels $M = 14$, and two different neural network architectures. That superiority was reinforced through experiments with varying workload types, values of N and M, and network architectures. Consequently, we adopt PPO as the algorithm for all subsequent evaluations.

The framework's default architecture is a multilayer perceptron (MLP) with two fully connected hidden layers of 64 neurons each (size

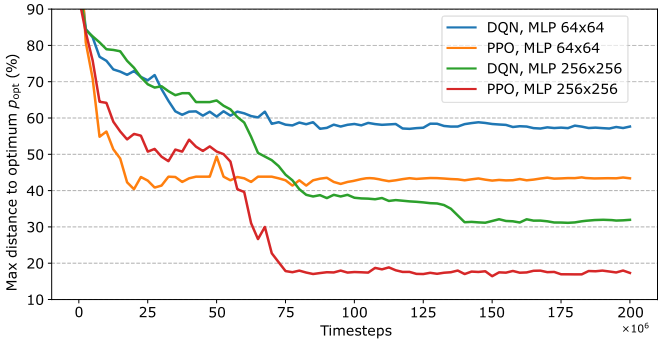

Fig. 10. Metric p_{opt} with 4 algorithm-architecture pairs, according to the number of training timesteps, for GOODSCALING datasets with $N = 14$ and $M = 14$.

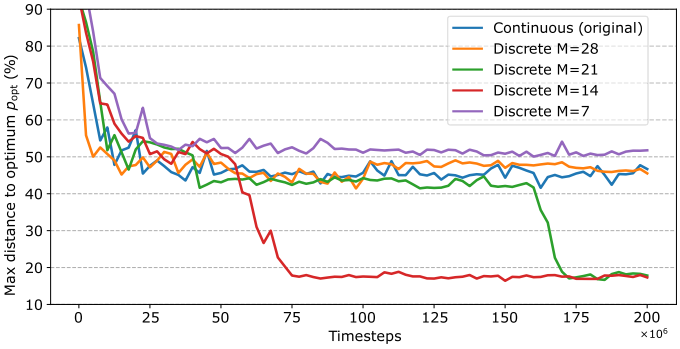

Fig. 11. Comparison of learning with continuous or discretized time representation with different granularity for $N = 14$ and GOODSCALING workloads.

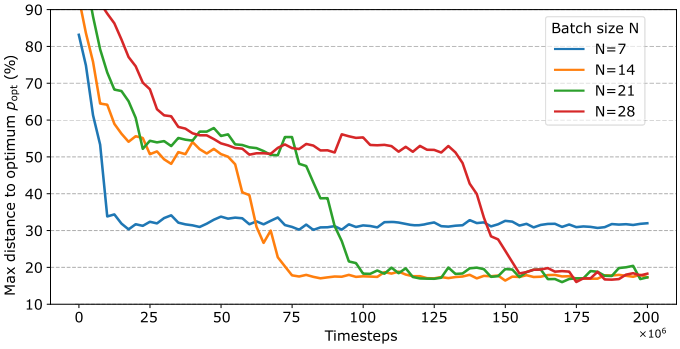

Fig. 12. Comparison of learning with different batch sizes N, using $M = 14$ and GOODSCALING workloads.

64×64). We modified it to a 256×256 configuration, which obtains better evaluation metric values p_{opt} within a reasonable training period, as also shown in Fig. 10. Small-scale tests with alternative architectures that scaled width and depth (such as $256 \times 256 \times 256$ and 512×512, among others) did not yield significant improvements and resulted in considerably longer training times. The final choice of architecture 256×256 will be further supported by the results obtained.

6.2. Time discretization

Regarding the discretization of task durations handled by the scheduler, we conducted several experiments comparing the use of original continuous time values with uniformly discretized representations of varying granularity. Specifically, the maximum representable time was divided into M evenly spaced levels, exploring different values of M.

Fig. 11 illustrates the evolution of the p_{opt} over 200 million training timesteps, comparing the agent's performance when using continuous time values versus discretized representations with $M \in \{7, 14, 21, 28\}$ levels. As mentioned above, p_{opt} includes the discretization error, so it is considered in the comparison. It begins with at least $M = 7$ levels since the GPU has that number of slices and it would be the minimum to represent the durations of a linear scaling task with different discrete times. The values of M progress in multiples of 7, gradually increasing the number of levels per slice for discretization. It is noteworthy that, even with a relatively large amount of training, learning with the original continuous time values has not yet achieved the improvements in p_{opt} observed in the discretized versions with $M = 14$ and $M = 21$. This observation, which also occurred for other tests with different batch sizes N, leads us to adopt the discretization. Additionally, $M = 14$ achieves a p_{opt} value similar to $M = 21$, but requires considerably fewer training timesteps and significantly less time (training is slower with higher M since the observation size, and consequently the number of neural network operations, depends on it). Therefore, we select $M = 14$ for subsequent evaluations.

It may be considered if the use of $M = 21$ or even $M = 28$ could be beneficial with substantially longer training. However, training must stop at some point (200 million timesteps was already considerable for us in this type of validation experiment), and at most 17% mean error versus the optimum seems difficult to be significantly improved. This choice for M will also be validated by the results of our approach, which will meet one of our main objectives by outperforming the baselines.

6.3. Batch size

Similarly, we search for an appropriate value for the hyperparameter N that defines the batch size. Given that a GPU with 7 slices can execute this number of tasks in parallel, we begin our experiments with $N = 7$ to ensure that the model processes information from all tasks that could potentially run concurrently. From there, we test at multiples of 7, obtaining during training the p_{opt} shown in Fig. 12, for GOODSCALING workloads and the $M = 14$ value selected previously. It can be observed that the value to which $N = 7$ converges after 200 training timesteps is inferior to that achieved with higher values of N, suggesting that 7 tasks are not enough. Higher values ($N \in \{14, 21, 28\}$) converge to approximately the same results, but require more iterations as N increases, with each training time step being slower as more operations are required. Consequently, we choose $N = 14$ since it converges to the same value as larger N but significantly faster.

6.4. Discrete time encoding

Based on the previous experiments, we decided to discretize the times for the representation, and we will now analyze two possible discrete encodings for the state. The first approach consists of a ONE-HOT encoding, i.e., a M-dimensional vector with all components fixed at 0, except the component corresponding to the assigned discretization level, which is set to 1. The alternative, which we call FLOAT representation, directly uses the nearest quantized real value after dividing the time into M equally spaced levels. For example, discretizing the interval [0,1] with $M = 4$ would have levels 0.25, 0.5, 0.75 and 1[9], so a value of 0.6 would be quantized to the nearest level, 0.5, with ONE-HOT encoding (0, 1, 0, 0) to indicate the second level, while its FLOAT representation would simply be 0.5[10].

On the one hand, ONE-HOT is widely used for categorical variables in machine learning and is the default for discrete representations in

[9] Note that there is no level for 0, because the model would interpret an instance as still free when assigning to it a task with duration 0. We reserve the value 0 to represent task completion and resource availability.

[10] Tasks that are not executable on certain instance size due to lack of resources take infinite time, and are not quantized in FLOAT, while are represented by an entire vector of −1 in ONE-HOT

Table 4
Average p_{opt} (%) with respect to the optimum for different real and synthetic workloads with $M = 14$ and $N = 14$ after 200 million training timesteps.

	Baseline				RL agent version			
Workload	FAR	MISO	FixBest	NoMIG	One-Hot	Float	Entropy	Direct Reconfig
Rodinia + Altis	22.71	26.72	75.37	101.52	89.53	89.36	70.22	21.39
PoorScaling	18.49	19.21	19.21	173.58	15.19	12.64	12.65	12.65
GoodScaling	18.68	20.39	23.38	25.17	17.41	14.68	14.94	14.89
MixScalingUniform	21.95	27.17	94.27	140.11	86.24	85.49	68.47	20.03
MixScalingExtreme	23.09	25.60	92.73	112.35	78.86	78.41	20.95	19.56
WideTimes	21.66	27.23	87.81	129.00	87.96	86.54	70.73	20.38

Synthetic workloads are trained by generating times of such type, and after evaluated on its 1000 reference datasets. Training for Rodinia + Altis is conducted with WideTimes workloads (the most general and well distributed synthetic ones), and is evaluated on the 1000 fixed subsets of real benchmarks.

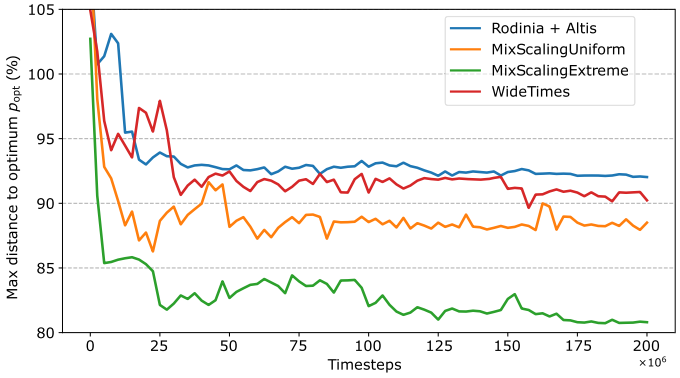

Fig. 13. Evolution of p_{opt} during the training of the RL agent with Float version for workloads exhibiting the no-reconfiguration issue.

the *Gymnasium* modeling library we use; the idea is that decomposing the levels into different components of the neural network's input can facilitate learning. However, the dimensionality of One-Hot encoding depends on M, resulting in $N \cdot M \cdot K$ inputs for Batch$_t$ to represent the K time instances of N entries in the batch, while the Float representation only requires $N \cdot K$ inputs. This dimensionality increase leads to a larger initial layer, thereby requiring more computations, slowing down training, and potentially extending convergence times. In cases where M is large, this can be counterproductive. To determine the best option for our scenario, we trained the agent using both methods for 200 million timesteps with previously selected values of N and M, and computed the p_{opt} metric for the different workloads. The results, shown in the One-Hot and Float columns of Table 4, reveal that the Float representation performed slightly better than One-Hot in all cases. In addition, each training timestep is faster with Float. Consequently, we adopt the Float representation hereafter.

6.5. Problem of non-reconfiguration

From the results for the One-Hot and Float versions, it is striking that the results for the Rodinia + Altis workloads, as well as for MixScalingUniform, MixScalingExtreme, and WideTimes, are considerably poor, yielding p_{opt} values exceeding 80% relative to the optimum (highlighted in red in Table 4). However, in those cases, the baselines reach a little more than 20% error. On the other hand, for the PoorScaling and GoodScaling workloads, the RL versions obtain highly competitive p_{opt} values, below the 20% distance to the optimum, outperforming the baselines.

The key difference between the mentioned workloads lies in the diversity of the scaling capacities of the tasks: whereas all PoorScaling tasks can only efficiently use a few slices, and all GoodScaling tasks can efficiently use many slices, the other workloads include a mix (some tasks scale well, while others do not). In these mixed scenarios, dynamic reconfiguration is critical, as different co-execution schemes will be required for tasks with such varied scaling characteristics. Conversely, when all tasks exhibit poor scaling (PoorScaling), employing the fixed configuration 1-1-1-1-1-1-1 typically yields the best results, while for workloads in which all tasks scale well (GoodScaling), using fixed configurations like 7 or 4-3 tends to be highly effective. Looking at concrete examples of problematic workloads, we found that the model tended to perform too few reconfiguration actions. Counting the number of reconfiguration actions taken during training revealed that these actions were hardly explored. Moreover, the learning curves in Fig. 13 demonstrated a very rapid convergence to what is probably a considerably suboptimal local minimum, from which the model could not escape.

6.5.1. Improvement 1: promote training exploration

As a result of discovering premature convergence and the agent's inability to escape local minima, we decided to significantly favor exploration over exploitation, during a considerable portion of the training timesteps. RL algorithms offer tunable hyperparameters to manage this balance; in the PPO algorithm, the entropy coefficient plays a key role by promoting model randomness and, consequently, exploration. By default, the coefficient was set to 0 (meaning no entropy influence), and by increasing it to 0.01 we substantially enhanced exploration without causing excessive randomness that would lead the model to unlearn (an issue observed with higher entropy values). Since this parameter value enabled the model to escape local minima, we implemented a milestone-based annealing: during the first quarter of the training timesteps the entropy was maintained at 0.01, then reduced to 0.075 for the subsequent quarter, further decreased to 0.005 for the third quarter, and finally set to 0 for the last quarter.

As reflected in the Entropy column of Table 4, there is a remarkable improvement for MixScalingExtreme workloads: from $p_{opt} = 77.2\%$ of Float, it pass to only 21.6%, surpassing the baselines (23.67% of FAR and 26.11% of MISO). However, despite slight improvements, for the remaining mixed workloads (MixScalingUniform, WideTimes, and Rodinia + Altis) performance remains poor, and is still significantly worse than baselines. We observed that the MixScalingExtreme workloads, which are polarized between tasks that scale very well and those that scale poorly, tended to plan effectively with a single reconfiguration from 1-1-1-1-1-1-1 to 7 or 4-3, a behavior already learned by the model. In contrast, the remaining mixed workloads require more frequent or varied reconfigurations, and the model still has difficulties learning those patterns.

6.5.2. Improvement 2: direct reconfiguration actions

Although the previous improvement encouraged the exploration of new actions, reconfigurations remained underutilized. Thus, we attempt to simplify the reconfiguration agent's decision by eliminating the precondition described in Section 4.2. Until now, reconfiguration was only

allowed if the slices to be modified from the current partition to the target partition were free. Although the model technically allows reconfiguration from any partition to any other partition by taking the *Advance* action enough times, the probability of sampling such a chain of actions during training is low. It is the conditional probability of multiple *Advance* actions until sufficient slices become available for a configuration change. Consequently, when GPU occupancy is high, it is very unlikely to transition to a configuration that differs substantially from the current one.

As reconfigurations are particularly useful for general workloads (including the RODINIA + ALTIS benchmarks), we modified their action requirements to allow switching to any partition in any state, with the necessary *Advance* actions implicitly executed rather than requiring the model to chain them probabilistically. It is similar to looking at several moves forward, which RL already handles through the value function and accumulated rewards, but for reconfiguration cases we deliberately promote that to accelerate its learning. This upgrade was not directly integrated into the original model, as it is derived from experimental observations in our specific scenario, in contrast with other optimizations discussed in Section 4. To simplify agent learning, we include an additional bit in the state indicating if the last action was a reconfiguration and disabling reconfigurations for that state. The new result of the action is:

DIRECT RECONFIG to C

Result: As in the original action, the destruction and creation times are added to the slices of those instances where the current configuration $\text{Canon}(\text{Conf}_t) = (c_1^t, \ldots, c_K^t)$ differs from the target configuration $C = (c_1, \ldots, c_K)$:

$$S_i^t.\text{time} \mathrel{+}= t_{\text{destroy}}(c_i^t) \quad \text{if } c_i^t \neq c_i,$$

$$S_i^t.\text{time} \mathrel{+}= t_{\text{create}}(c_i) \quad \text{if } c_i^t \neq c_i,$$

[11] but then the GPU time is advanced until all these modifications are complete with the corresponding reward:

$$S_i^{t+1}.\text{time} \mathrel{-}= \text{reconfig\_delay} \quad \forall\, 1 \leq i \leq K,$$

$$r_t = -\text{reconfig\_delay},$$

$$\text{with reconfig\_delay} = \max\left\{S_i^{t+1}.\text{time} : c_i^t \neq c_i,\ 1 \leq i \leq K\right\}.$$

With this new action, the GPU at timestep t of Fig. 6 could be directly reconfigured to 4-3 with the result shown in Fig. 14. An implicit *Advance* of 4 time units has occurred to free instances $\{S_1, S_2\}$ and $\{S_3\}$, merging them with $\{S_4\}$, which was already free. The task of class C_1 is then assigned to the new instance of size 4, which scales very efficiently at that size. The outcome at timestep $t+2$, also shown in Fig. 14, is significantly better to the decisions depicted in Fig. 6. In fact, the decision will be much better than that, because the rest of the tasks to be placed (those of class C_2) also scale very well to 4 slices, so it is ideal to have transitioned 4-3. Under the original modeling, two consecutive *Advance* actions would have been necessary before reconfiguring to 4-3, which would have been highly unlikely to sample during training.

The column DIRECT RECONFIG of Table 4 reports the results of this enhanced agent version in different workloads. In cases where the agent did not previously reconfigure in an effective way, the improvement is substantial, with values of p_{opt} now about 20%. These results clearly demonstrate the strong performance of the model, which is remarkably close to the optimal solution, even better in practice than that percentage indicates. Furthermore, this approach consistently outperforms all baselines: it achieves a slight improvement over our previous method FAR, a more noticeable gain over the state-of-the-art approach MISO, and a significant advantage over the methods without dynamic reconfiguration (FIXBEST) or without co-execution (NOMIG).

[11] Obviously the reconfiguration times t_{destroy} and t_{create} are scaled according to task times discretization.

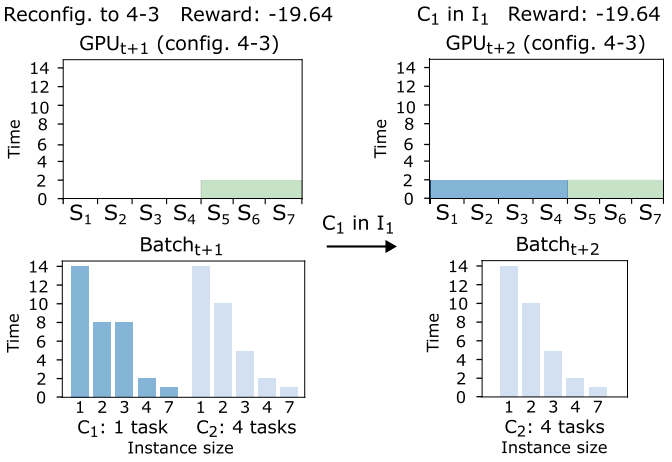

Fig. 14. Timestep $t+1$ shows the state of the scheduler after reconfiguring directly to 4-3 from timestep t of Fig. 6. The timestep $t+2$ shows the result of allocating the task of class C_1 in the new instance of size 4.

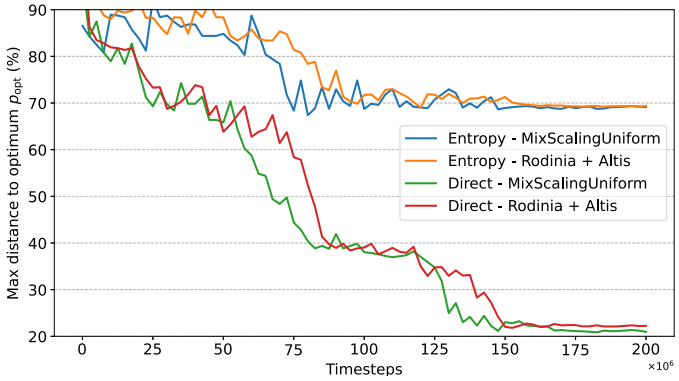

Fig. 15. Learning curves of the ENTROPY's agent and the version also including DIRECT RECONFIG, for workloads with the non-reconfiguration problem.

Fig. 15 shows the training curves of the previous model version (ENTROPY), versus the one with the new reconfiguration actions (DIRECT RECONFIG), for two workloads where the non-reconfiguration problem were unresolved (RODINIA + ALTIS and MIXSCALINGUNIFORM). The benefits of new actions are evident, both in terms of higher reward and faster convergence. Around the midpoint of training, the availability of direct reconfiguration enables the agent to reduce the distance to the optimum from $p_{\text{opt}} \sim 70\%$ to $\sim 40\%$—while ENTROPY remains stagnant near to 70%. As training progresses, just before reaching the final quarter, DIRECT RECONFIG achieves further improvement, lowering to $p_{\text{opt}} \sim 20\%$, but ENTROPY continues at 70%.

6.6. Online approach

Finally, we evaluate the results of the online approach, whose advantages and disadvantages compared to the offline method were discussed in Section 3. Table 5 shows the percentage improvement p^A_{online} of the refined DIRECT RECONFIG version of the online RL agent, compared to the state-of-the-art baselines and the same agent's version of the offline approach. It should be noted that this agent was trained for more timesteps than the offline versions in Table 4 (500 million vs. 200 million) because the same number of timesteps did not produce sufficiently good results. This is probably due to its stochastic nature, which slows learning but brings more richness to the scheduling process. To ensure a fair comparison between the online and offline approaches, the offline DIRECT RECONFIG method was also trained for 500 million timesteps for Table 5, with no noticeable gain compared to 200 million timesteps.

Table 5
Percentage improvement p^{A}_{online} (%) of the online DIRECT RECONFIG after 500 millions of training timesteps vs. approaches A in the columns.

	Baseline		Offline RL agent
Workload	FAR	MISO	DIRECT RECONFIG
RODINIA + ALTIS	2.45	5.80	1.35
POORSCALING	4.21	4.85	−0.92
GOODSCALING	2.55	4.02	−0.73
MIXSCALINGUNIFORM	2.78	7.18	1.16
MIXSCALINGEXTREME	3.28	5.39	0.32
WIDETIMES	2.31	7.01	1.25

As Table 5 shows, after 500 million timesteps, the online approach slightly improves the makespan of the offline method by approximately 1% for workloads that are varied or exhibit heterogeneous durations (RODINIA + ALTIS, MIXSCALINGUNIFORM, MIXSCALINGEXTREME, and WIDETIMES). These were precisely the workloads where the offline approach's improvement over the baselines was less pronounced and further from the optimum, as observed in Table 4. However, for the more homogeneous workloads, POORSCALING and GOODSCALING, the online approach performed slightly worse than the offline one (by around 1%). As consequence, the results against baselines are similar with different types of workloads, and in some cases even better for varied tasks of heterogeneous duration; improvements range between 2% and 4% against FAR and between 4% and 7% against MISO.

For both approaches, it is important to consider the overhead resulting from the scheduler's inference latency. As seen in Section 6.1, very large MLP architectures are not needed to obtain good results. In particular, over a sample of 1000 consecutive inferences, an average latency of only 3.04 ms was obtained for the final model (MLP 256×256 with $N = M = 14$), with a 99th-percentile of 3.09 ms, running on a server with an Intel Xeon Silver 4314 CPU (16 cores @2.40GHz). These results guarantee the viability of the proposal in a real online environment, for all classes of tasks (even very short tasks).

6.7. Assessment of results

Our final RL model (DIRECT RECONFIG) achieves improvements of 2–7% across all workload types compared to state-of-the-art baselines. Although the improvements are moderate, it should be noted that FAR—the best performing baseline—is a very recent proposal, quite refined and specifically aimed at minimizing makespan under current MIG conditions, which contrasts with the flexibility in objective and model specification offered by our DRL proposal. While previous heuristic-based proposals require a complete redesign to change the optimization objective, DRL requires only a simple change of the reward function and retraining of the model. Similarly, if the characteristics of the workloads change over time (*data drift*), DRL simply requires retraining with new inputs, whereas previous methods would need extensive rethinking to adapt them to the new features. In addition to all these advantages, our proposal also achieves a sustained improvement in quite varied and general scenarios compared to the recent state-of-the-art.

As motivated in Section 2.2, these gains are substantially greater when the simplifications of prior approaches are untenable. To evaluate the merits of our approach in more specific scenarios where these problems are exacerbated, we generate 1000 new datasets using the synthetic task generator described in Section 5.1.2. Specifically, we reflect two situations for NVIDIA A100 where assigning each task to its optimal efficiency size is detrimental in terms of their joint execution:

1. 500 datasets containing only tasks whose peak efficiency occurs at 4 slices, but whose efficiency at 3 slices is only marginally worse. In that case, a uniform 4–4 configuration is infeasible (see Fig. 1), so individual optimization would prevent any co-execution. By contrast, a mixed 4–3 allocation allows concurrent execution even if it is not individually optimal for 3-slice tasks.
2. 500 datasets containing only tasks whose peak efficiency occurs at 2 slices, but is not much worse with 3 slices. However, a 2–2–3 configuration generally yields better makespan, since the third task in the configuration is accelerated by a factor near to 3/2 avoiding idle resources.

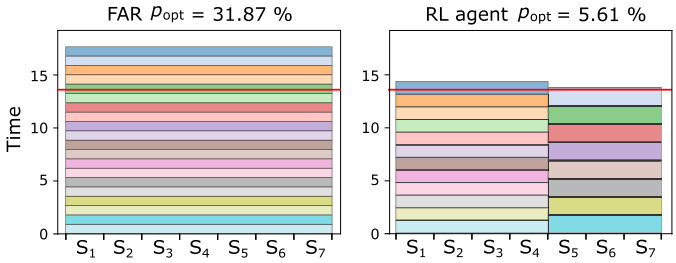

Fig. 16. Schedule of FAR vs. DIRECT RECONFIG RL agent for 20 repetitions of the Rodinia's *CFD* benchmark.

On these workloads, FAR achieves $p_{opt} = 27.82\%$, MISO 41.23%, and our DRL method 18.10%—which translates into makespan reductions of 8.23% and 19.59%, respectively.[12] In particular, Fig. 16 illustrates the schedules produced by FAR and our DRL agent for a batch of 20 runs of the Rodinia CFD benchmark. Although the benchmark's individual optimum is at 4 slices, the best co-execution pattern is 4–3-exactly the configuration discovered by our agent. FAR, however, has mechanisms to find an assignment that exceeds the uniform 4-slice assignment, assigning 7 slices to each task, but falls far short of the 4-3 pattern. In this experiment, FAR records $p_{opt} = 31.87\%$ versus 5.61% of our method, corresponding to a 24.87% makespan improvement. While this is a pathological situation, it illustrates problems that the agent solves in more general situations, albeit less frequently, as a result of avoiding the assumptions of other proposals.

7. Conclusions and learned lessons

As a result of the design and evaluation of our model, several conclusions emerge: **Feasibility and benefits of a holistic approach** It was shown that a complex scheduling problem, until now approached by optimizing its dimensions separately (such as task molding independently from reconfiguration and co-execution decisions), can be approached jointly despite its vast theoretical amount of possibilities. Deep Reinforcement Learning, with its efficient exploration and sampling techniques, enables us to consider all dimensions simultaneously without oversimplification. As shown in Tables 4 and 5, the final versions of our RL agent achieve a 2%-7% makespan reduction versus the state-of-the-art, with an average deviation from the optimal schedule of no more than approximately 20% (in fact, even lower). The improvements are especially significant in scenarios that exacerbate problems related to simplifications made in the state of the art, such as those presented in Section 6.7, reducing the makespan by 8%–20%. **Importance of design and representation** When facing a problem of such high dimensionality, it is crucial to provide the agent with an expressive, but also simple and abstract, representation. Our experience shows that reducing unnecessary cardinality and facilitating rapid exploration of key scenarios is paramount. In this regard, the following refinements were made:

- Discretizing task times with an adjustable parameter M.
- Eliminating useless configurations that reserve four slices while only using three.

[12] Note that the makespan improvement cannot be computed directly from $p_{opt} = (\text{makespan} / l\_bound - 1) \cdot 100$; instead, one must first solve for the makespan itself and then compare those values.

- Establishing a canonical and compact representation of batched tasks by sorting and grouping repeated tasks.
- Unify equivalent configurations that formed isomorphic symmetries in the states.
- Explicitly masking prohibited actions in the state instead of relying on the model to learn to avoid them.
- Explore FLOAT encoding for discrete state values as an alternative to ONE-HOT.
- Promote reconfiguration after experimentally observing their under-exploration in critical cases. As we saw in Fig. 15, DIRECT RECONFIG improves ∼50% for mixed workloads with the same amount of training.

Although some of these techniques may not be useful in other works, the key conclusion is that investing effort in refining the state representation and evaluating alternative approaches can be very beneficial.

Relevance of the training setup Although it is often necessary to limit the number of test configurations, choosing the right training algorithm, model architecture and hyperparameters are fundamental. We explored a variety of alternatives, as detailed in Sections 6.1, 6.2 and 6.3. In our case, the adoption of the PPO algorithm over DQN and the use of an MLP architecture 256×256 provided marked improvements. Also, as shown in Fig. 11, $M = 14$ discretization levels was much better than finer or coarser granularities, and than non-discretization.

Synthetic data and validation with real data In scenarios where real datasets are insufficient for machine learning, it is essential to generate plausible synthetic cases, based on general patterns extracted from real cases and domain knowledge, as in Section 5.1.2. It is also important to validate the model by evaluating on real cases, ideally unknown for training, as in Table 4 with the subsets of RODINIA + ALTIS, obtaining results similar to the synthetic tests. To obtain more disaggregated results and detect problems, it is useful to train and evaluate in isolation with well-differentiated and distributed workload types, such as those presented in Section 5.1.3. For example, isolating datasets with homogeneous workloads from datasets with mixed workloads made it easier to discover the problem of non-reconfiguration (Section 6.5), since it only occurred in the mixed ones.

CRediT authorship contribution statement

Jorge Villarrubia: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Formal analysis, Conceptualization; **Luis Costero:** Writing – review & editing, Validation, Supervision, Formal analysis, Conceptualization; **Francisco D. Igual:** Writing – review & editing, Validation, Supervision, Formal analysis, Conceptualization; **Katzalin Olcoz:** Writing – review & editing, Validation, Supervision, Conceptualization.

Data availability

The associated tools can be found in the public repository https://github.com/artecs-group/RL_MIG_scheduler.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Jorge Villarrubia, Luis Costero, Francisco D. Igual, Katzalin Olcoz reports financial support was provided by Spain Ministry of Science and Innovation. If there are other authors, they 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 and Code Availability

This work is funded by Grant PID2021-126576NB-I00 funded by MCIN/AEI/10.13039/501100011033 and by *"ERDF A way of making Europe"*. We thank the HPC&A group at Universitat Jaume I for granting us access to A100 and H100 GPUs.

The implementation of the model trainer, the schedule visualizer and the MIG co-execution tool on GPU is publicly available under the MIT license at [22].

References

[1] T. Adufu, J. Ha, Y. Kim, Exploring the Diversity of Multiple Job Deployments over GPUs for Efficient Resource Sharing, in: International Conference on Information Networking, IEEE Computer Society, 2024, pp. 777–782. https://doi.org/10.1109/ICOIN59985.2024.10572198
[2] J. Villarrubia, L. Costero, F.D. Igual, K. Olcoz, Leveraging Multi-Instance GPUs through moldable task scheduling, J. Parallel Distrib. Comput. 204 (2025) 105128. https://doi.org/10.1016/j.jpdc.2025.105128
[3] C. Espenshade, R. Peng, E. Hong, M. Calman, Y. Zhu, P. Parida, E.K. Lee, M.A. Kim, Characterizing training performance and energy for foundation models and image classifiers on multi-instance GPUs, in: 4th Workshop on Machine Learning and Systems, ACM, 2024, pp. 47–55. https://doi.org/10.1145/3642970.3655830
[4] B. Li, V. Gadepally, S. Samsi, D. Tiwari, Characterizing Multi-Instance GPU for Machine Learning Workloads, in: IEEE 36th International Parallel and Distributed Processing Symposium Workshops, 2022. https://doi.org/10.1109/IPDPSW55747.2022.00124
[5] A. Jahanshahi, M. Rezvani, D. Wong, WattWiser: power & resource-efficient scheduling for multi-model multi-GPU inference servers, in: 14th International Green and Sustainable Computing Conference, 2024, p. 39–44. https://doi.org/10.1145/3634769.3634807
[6] NVIDIA, Multi-Process Service, (URL https://docs.nvidia.com/deploy/mps/).
[7] NVIDIA, Multi-Instance GPU User Guide, (URL https://docs.nvidia.com/datacenter/tesla/mig-user-guide/).
[8] B. Zhang, S. Li, Z. Li, MIGER: Integrating Multi-Instance GPU and Multi-Process Service for Deep Learning Clusters, in: 53rd International Conference on Parallel Processing, 2024, p. 504–513. https://doi.org/10.1145/3673038.3673089
[9] B. Li, T. Patel, S. Samsi, V. Gadepally, D. Tiwari, MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters, in: 13th Symposium on Cloud Computing, 2022. https://doi.org/10.1145/3542929.3563510
[10] AMD, MxGPU Deployment Guide, (URL https://drivers.amd.com/relnotes/amd_mxgpu_deploymentguide_vmware.pdf).
[11] SchedMD, SLURM documentation: MIG management, (URL https://slurm.schedmd.com/gres.html#MIG_Management).
[12] Center for High Throughput Computing,, U. o. W.-M. HTCondor User Manual.
[13] NVIDIA, GPU Operator: Use of MIG via Kubernetes, (URL https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-operator-mig.html).
[14] J. Du, J.Y.T. Leung, Complexity of Scheduling Parallel Task Systems, SIAM Journal on Discrete Mathematics 2 (4) (1989) 473–487. https://doi.org/10.1137/0402042
[15] M. Lee, S. Seong, M. Kang, J. Lee, G.-J. Na, I.-G. Chun, D. Nikolopoulos, C.-H. Hong, ParvaGPU: efficient spatial GPU sharing for large-scale DNN inference in cloud environments, in: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2024, pp. 1–14. https://doi.org/10.1109/SC41406.2024.00048
[16] J. Qi, W. Xiao, M. Li, C. Yang, Y. Li, W. Lin, H. Yang, Z. Luan, D. Qian, ElasticBatch: A Learning-Augmented Elastic Scheduling System for Batch Inference on MIG, IEEE Trans. Parallel Distrib. Syst. (2024). https://doi.org/10.1109/TPDS.2024.3431189
[17] B. Turkkan, P. Murali, P. Harsha, R. Arora, G. Vanloo, C. Narayanaswami, Optimal Workload Placement on Multi-Instance GPUs, 2024, arXiv:2409.06646
[18] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, A Bradford Book, Cambridge, MA, USA, Cambridge, MA, USA, 2018.
[19] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al., A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play, Science 362 (6419) (2018) 1140–1144. https://doi.org/10.1126/science.aar640
[20] Y. Gu, Z. Liu, S. Dai, C. Liu, Y. Wang, S. Wang, G. Theodoropoulos, L. Cheng, Deep Reinforcement Learning for Job Scheduling and Resource Management in Cloud Computing: An Algorithm-Level Review, 2025, arXiv:2501.01007
[21] X. Wang, S. Wang, X. Liang, D. Zhao, J. Huang, X. Xu, B. Dai, Q. Miao, Deep Reinforcement Learning: A Survey, IEEE Trans. Neural Netw. Learn. Syst. 35 (4) (2024) 5064–5078. https://doi.org/10.1109/TNNLS.2022.3207346
[22] J. Villarrubia, Code repository: RL MIG scheduler, 2025, https://github.com/artecs-group/RL_MIG_scheduler.
[23] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.-H. Lee, K. Skadron, Rodinia: A benchmark suite for heterogeneous computing, in: 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 44–54. https://doi.org/10.1109/IISWC.2009.5306797
[24] B. Hu, C.J. Rossbach, Altis: modernizing GPGPU benchmarks, in: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2020, pp. 1–11. https://doi.org/10.1109/ISPASS48437.2020.00011
[25] Y. Kim, Y. Choi, M. Rhu, PARIS and ELSA: an elastic scheduling algorithm for reconfigurable multi-GPU inference servers, in: Design Automation Conference, 2022. https://doi.org/10.1145/3489517.3530510
[26] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, J. Huh, Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing, in: USENIX ATC, 2022.

[27] X. Wei, Z. Li, C. Tan, Optimizing GPU sharing for container-based DNN serving with multi-instance GPUs, in: 17th ACM International Systems and Storage Conference, 2024, pp. 68–82. https://doi.org/10.1145/3688351.3689156
[28] Project V., Volcano documentation of dynamic-MIG feature, (URL https://github.com/volcano-sh/volcano/blob/master/docs/design/dynamic-mig.md).
[29] S.F. Apache, Yunikorn user guide for NVIDIA GPUs, (https://yunikorn.apache.org/docs/user_guide/workloads/run_nvidia/).
[30] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, D. Trystram, Scheduling Independent Moldable Tasks on Multi-Cores with GPUs, IEEE Trans. Parallel Distrib. Syst. 28 (9) (2017) 2689–2702. https://doi.org/10.1109/TPDS.2017.2675891
[31] X. Wu, P. Loiseau, Efficient approximation algorithms for scheduling moldable tasks, Eur. J. Oper. Res. 310 (1) (2023) 71–83. https://doi.org/10.1016/j.ejor.2023.02.044
[32] B.M. Kayhan, G. Yildiz, Reinforcement learning applications to machine scheduling problems: a comprehensive literature review, 2023, https://doi.org/10.1007/s10845-021-01847-3
[33] H. Mao, M. Alizadeh, I. Menache, S. Kandula, Resource management with deep reinforcement learning, in: HotNets - 15th ACM Workshop on Hot Topics in Networks, 2016. https://doi.org/10.1145/3005745.3005750
[34] C. Shyalika, T. Silva, A. Karunananda, Reinforcement Learning in Dynamic Task Scheduling: A Review, 2020, https://doi.org/10.1007/s42979-020-00326-5
[35] D. Qiao, L. Duan, H.L. Li, Y. Xiao, Optimization of job shop scheduling problem based on deep reinforcement learning, Evolutionary Intelligence 17 (2024). https://doi.org/10.1007/s12065-023-00885-5
[36] T. Zhou, H. Zhu, D. Tang, C. Liu, Q. Cai, W. Shi, Y. Gui, Reinforcement learning for online optimization of job-shop scheduling in a smart manufacturing factory, Advances in Mechanical Engineering 14 (3) (2022). https://doi.org/10.1177/16878132221086120
[37] C. Perez Bernal, M.A. Salido, C. March Moya, Optimizing energy efficiency in unrelated parallel machine scheduling problem through reinforcement learning, Inf. Sci. 693 (2025) 121674. https://doi.org/10.1016/j.ins.2024.121674
[38] Y. Fan, Z. Lan, T. Childers, P. Rich, W. Allcock, M.E. Papka, Deep reinforcement agent for scheduling in HPC, in: IEEE 35th Int. Parallel and Distributed Processing Symposium, 2021. https://doi.org/10.1109/IPDPS49936.2021.00090
[39] J. Fomperosa, M. Ibañez, E. Stafford, J.L. Bosque, Task scheduler for heterogeneous data centres based on deep reinforcement learning, in: Parallel Processing and Applied Mathematics, 2023, pp. 237–248. https://doi.org/10.1007/978-3-031-30442-2_18
[40] B. Li, Y. Fan, M. Dearing, Z. Lan, P. Rich, W. Allcock, M. Papka, MRSch: multi-resource scheduling for HPC, in: IEEE International Conference on Cluster Computing, 2022. https://doi.org/10.1109/CLUSTER51413.2022.00020
[41] W. Lie, W. Feng-yan, Dynamic partial reconfiguration in FPGAs, in: Third International Symposium on Intelligent Information Technology Application, 2, 2009, pp. 445–448. https://doi.org/10.1109/IITA.2009.334
[42] A. Dhar, E. Richter, M. Yu, W. Zuo, X. Wang, N.S. Kim, D. Chen, DML: Dynamic Partial Reconfiguration with Scalable Task Scheduling for Multi-Applications on FPGAs, IEEE Transactions on Computers 71 (2022). https://doi.org/10.1109/TC.2021.3137785
[43] Q. Tang, Z. Wang, B. Guo, L.-H. Zhu, J.-B. Wei, Partitioning and Scheduling with Module Merging on Dynamic Partial Reconfigurable FPGAs, ACM Trans. Reconfigurable Technol. Syst. 13 (3) (2020). https://doi.org/10.1145/3403702
[44] A. Ben-Ameur, A. Araldo, T. Chahed, G. Dán, Cache Allocation in Multi-tenant Edge Computing: an Online Model-based Reinforcement Learning Approach, IEEE Trans. Cloud Comput. (2025) 1–14. https://doi.org/10.1109/TCC.2025.3538158
[45] R. Jia, Z. Chen, C. Wu, J. Li, M. Guo, H. Huang, RL-Cache: an efficient reinforcement learning based cache partitioning approach for multi-tenant CDN services, in: IEEE International Conference on Cluster Computing, 2024, pp. 202–213. https://doi.org/10.1109/CLUSTER59578.2024.00025
[46] R. Si, S. Chen, J. Zhang, J. Xu, L. Zhang, A multi-agent reinforcement learning method for distribution system restoration considering dynamic network reconfiguration, Appl. Energy 372 (2024) 123625. https://doi.org/10.1016/j.apenergy.2024.123625
[47] H. Mao, M. Schwarzkopf, S.B. Venkatakrishnan, Z. Meng, M. Alizadeh, Learning scheduling algorithms for data processing clusters, in: SIGCOMM - Conference of the ACM Special Interest Group on Data Communication, 2019. https://doi.org/10.1145/3341302.3342080
[48] F. Saeik, M. Avgeris, D. Spatharakis, N. Santi, D. Dechouniotis, J. Violos, A. Leivadeas, N. Athanasopoulos, N. Mitton, S. Papavassiliou, Task offloading in Edge and Cloud Computing: A survey on mathematical, artificial intelligence and control theory solutions, Comput. Netw. 195 (2021) 108177. https://doi.org/10.1016/j.comnet.2021.108177
[49] O. Abraham, M. Asri Bin Ngadi, J. Bin Mohamad, S. Kufaisal Mohd, Task Scheduling in Cloud Environment-Techniques, Applications, and Tools: A Systematic Literature Review, IEEE Access 12 (2024) 138252–138279. https://doi.org/10.1109/ACCESS.2024.3466529
[50] M. Khadivi, T. Charter, M. Yaghoubi, M. Jalayer, M. Ahang, A. Shojaeinasab, H. Najjaran, Deep reinforcement learning for machine scheduling: Methodology, the state-of-the-art, and future directions, Comput. Ind. Eng. 200 (2025) 110856. https://doi.org/10.1016/j.cie.2025.110856
[51] V. Kumar, S. Bhambri, P.G. Shambharkar, Multiple Resource Management and Burst Time Prediction using Deep Reinforcement Learning, Eighth Intl. Conf. Adv. Comput. Commun. Inf. Technol. (2019). https://doi.org/10.15224/978-1-63248-169-6-09
[52] S.R. Sinclair, S. Banerjee, C.L. Yu, Adaptive Discretization in Online Reinforcement Learning, Oper. Res. (2023). https://doi.org/10.1287/opre.2022.2396
[53] S. Huang, S. Ontañón, A Closer Look at Invalid Action Masking in Policy Gradient Algorithms, The Intl. FLAIRS Conference Proceedings 35 (2022). https://doi.org/10.32473/flairs.v35i.130584
[54] R. Stolz, H. Krasowski, J. Thumm, M. Eichelbeck, P. Gassert, M. Althoff, Excluding the Irrelevant: Focusing Reinforcement Learning through Continuous Action Masking, 2024, arXiv:2406.03704
[55] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, Stable-Baselines3: Reliable Reinforcement Learning Implementations, J. Mach. Learn. Res. 22 (268) (2021) 1–8.
[56] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, K. Goldberg, J. Gonzalez, M. Jordan, I. Stoica, RLlib: abstractions for distributed reinforcement learning, in: 35th International Conference on Machine Learning, 80, PMLR, 2018, pp. 3053–3062.
[57] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing Atari with Deep Reinforcement Learning, 2013, arXiv:1312.5602
[58] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal Policy Optimization Algorithms, 2017, arXiv preprint arXiv:1707.06347