SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Reinforcement Learning based Resource Management for HPC Systems

Urvij Saroliya

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Reinforcement Learning based Resource Management for HPC Systems

Reinforcement Learning basierte Ressourcenmanagement für HPC-Systeme

Author:	Urvij Saroliya
Supervisor:	Prof. Dr. Martin Schulz
Advisor(s):	Dr. Eishi Arima, M.Sc. Dai Liu
Submission Date:	15.12.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, 15.12.2023 Urvij Saroliya

Acknowledgments

First and foremost, I wish to express my deepest gratitude to Dr. Eishi Arima for providing me with the remarkable opportunity to contribute to this project. His invaluable guidance and insightful feedback have been instrumental throughout the entire research process, shaping the direction and enhancing the quality of this thesis.

I would also like to extend gratitude to Dai Liu for her time and constructive feedback, which significantly contributed to refining the methodology of this thesis. I also want to sincerely thank Prof. Dr. Martin Schulz for his supervision and support, which was essential for the successful completion of this work.

I am profoundly thankful to my friends and family for their encouragement and understanding throughout the academic journey. In particular, I extend a heartfelt note of gratitude to my parents, whose unwavering support has been a constant and motivating force.

This work has received funding from the REGALE project from EuroHPC JU under grant agreement no. 956560 and the German Federal Ministry of Education and Research (BMBF) under grant number 16HPC039K. Further, it was supported by BMBF through the initiative SCALEXA and the PDExa project (16ME0641), and by the NVIDIA Academic Hardware Grant Program. Last but not the least, I would like to thank professors and staffs in SRD, ITC, The University of Tokyo, in particular Prof. Toshihiro Hanawa, for giving us access to their GPUs.

Abstract

In recent years there have been an exponential rise in the capabilities of the modern High Performance Computing (HPC) systems. Such trend poses new challenges for managing node-level resources such as compute cores, memory bandwidth, and shared cache. This has led to an increasing demand for effective resource management methodologies in HPC systems. As modern HPC systems are typically composed of fat and rich compute nodes, it is usually difficult to fully utilize all the in-node resources by a single application. Co-scheduling, i.e., co-locating multiple jobs in a space shared manner, offers a promising solution for improving overall system throughput. To this end, it is crucial to allocate the node resources to specific jobs based on their requirements. At the same time, during co-scheduling of multiple jobs, there is a further increase in the interference for the shared resources. Therefore, the significance of shared resource isolation increases during the allocation of resources to the co-located jobs. Furthermore, there have been a rise in heterogeneity of the node-level resources. GPU-based HPC systems are increasingly prevalent among top supercomputers. Hence, similar challenges are applicable to the GPU-based systems as well.

Considering these trends, industry has started supporting several resource partitioning or isolation features designed for shared resources on both modern CPUs and GPUs. Driven by this technological trend, we focus on co-scheduling and resource partitioning on modern CPU-GPU HPC systems. Specifically, for CPUs, our target is to harmonize the co-run job selections and diverse resource assignments in a NUMA-aware manner. Regarding GPUs, we explore hierarchical resource partitioning on latest NVIDIA GPUs, employing both finer-grained logical partitioning (MPS) and coarse-grained physical partitioning (MIG). To optimize resource management decisions, we implement a reinforcement learning-based solution, addressing CPU and GPU optimizations separately. Experimental evaluations demonstrates that our approach can improve the overall system throughput by up to 78.1% and 87.3% for CPU and GPU, respectively.

Contents

1 Introduction

Modern HPC clusters and supercomputers have seen an extraordinary surge in capabilities. Over the past decade alone, there has been more than 35-fold increase in the computational performance of top supercomputer, measured in terms of flops/s [82]. Ever since the end of Dennard scaling [25], these performance improvements have been driven by the adoption of multi-/many-core parallelism and heterogeneous architectures focusing on thread-/data-level parallelism. Consecutively, the industry has consistently increased the on-chip core counts, resulting in the widespread integration of many-core processors within HPC systems. Moreover, contemporary HPC systems are becoming increasingly heterogeneous, reflected in the fact that, as of November 2023, 187 out of the top 500 supercomputers are equipped with GPUs [82].

With such increased capabilities, the node-level resources have increased drastically, leading to fat and rich compute nodes. As a consequence, it is usually difficult to fully utilize all the in-node resources by a single application. Co-scheduling, which involves concurrently placing multiple applications through space sharing, stands out as a promising solution for enhancing overall system throughput. While co-scheduling contributes to improved performance, it also causes contention for shared resources such as cache and memory bandwidth. As a result, shared resource partitioning or isolation techniques are increasingly important during co-scheduling.

Industry has started supporting several resource/traffic partitioning features. For CPU, recent commercial microprocessors offer new hardware features designed for partitioning resources and traffic. Notable examples include cache and bandwidth partitioning features like Intel CAT/MBA, which are integral components of Intel Resource Director Technology [32]. Similarly, for modern accelerators like recent NVIDIA GPUs, multiple resource partitioning features are available, including: (1) MPS (Multi-Process Service) which enables the *logical* sharing of compute resources among multiple programs [52]; (2) MIG (Multi-Instance GPU) that has capabilities to *physically* partition compute and bandwidth resources at the granularity of the GPC [55].

To realize performance improvements through *co-scheduling* and the mentioned *resource partitioning* features, it is crucial to carefully apply optimal policies when scheduling jobs on an HPC system. In other words, this involves selecting complementary jobs for co-location and determining suitable resource allocations for these co-located jobs. As a consequence, an efficient algorithm becomes imperative for making informed scheduling decisions. This necessity led us to formulate the described *resource management* challenge as an optimization problem.

Reinforcement Learning is a fundamental machine learning paradigm that involves undertaking a series of actions in an environment to maximize the cumulative rewards. By leveraging these rewards, an agent can be trained to determine an optimal set of actions. This form of learning has demonstrated to perform well in various domains, including robotics, game-playing, autonomous vehicles, and natural language processing. In alignment with this trend, we employ a reinforcement learning-based solution to derive the optimal set of scheduling decisions for the resource management problem.

To enhance the computational throughput and memory bandwidth at the CPU level, contemporary nodes are often configured with multiple processor sockets, leading to inherently *Non-Uniform Memory Access* (NUMA) based designs [38]. This rationale guides us to utilize NUMA-based CPU in this work. Moreover, we use a recent NVIDIA GPU, leveraging its enhanced capabilities and essential partitioning features. Using these selected platforms, we conduct a comprehensive evaluation of our reinforcement learning-based approach, including comparisons with existing state-of-the-art methods wherever feasible.

1.1 Organization

This thesis has been organized into six additional chapters, each dedicated to describe the following aspects:

- **Background and Related Work:** In this chapter, we provide essential background and review existing related work to facilitate a thorough understanding of this work. This includes foundational details on profiling tools, co-scheduling, resource partitioning features, reinforcement learning, and neural networks.

- **Problem Definition and Solution Blueprint:** In this chapter, we initially present the mathematical formulation of the optimization problem. Subsequently, we provide a high-level blueprint of the proposed solution.

- **Harmonized Resource Management on NUMA Systems:** This chapter provides in-depth insights into observations, implementation details, and experimental evaluations, providing a focused examination on NUMA systems.

- **Hierarchical Resource Management on Modern GPUs:** This chapter provides comprehensive insights into observations, implementation details, and experimental evaluations, offering a focused examination on modern GPUs.

- **Discussion:** In this chapter, we engage in additional discussions on topics such as potential future opportunities to extend our approach, and comparison of our approach to other existing methods.
- **Conclusion:** Finally, this chapter conclude this thesis with a summary, highlighting the key takeaways.

1.2 Major Contributions

In this work, our approach is applied distinctly to both CPU and GPU. Subsequently, the key contributions for CPU and GPU are listed separately in the following sections.

1.2.1 NUMA Systems

1. We initially establish the correlation between core/memory mapping affinities and cache/bandwidth partitioning configurations on a NUMA-based platform.
2. Simultaneously, we observe that the selection of co-run job pairs significantly influences the effectiveness of both co-scheduling and partitioning decisions.
3. We apply our reinforcement learning-based systematic approach to solve the optimization problem on a NUMA system.
4. We ultimately quantify the effectiveness of our approach, demonstrating an substantial improvement of up to 78.1% in system performance compared to the time-shared scheduling.
5. Also, we provide comparison with another existing work done by Saba et el. [71].

1.2.2 Modern GPUs

1. This study marks the pioneering application of reinforcement learning to concurrently optimize job co-scheduling and hierarchical resource partitioning on modern GPUs, which incorporate multiple partitioning features (MPS and MIG).
2. We measure and analyze the impact of the various resource partitioning setups on the overall throughput of the GPU.
3. We experiment with hierarchical mixing of finer-grained MPS and coarse-grained MIG features.

4. Subsequently, we apply the reinforcement learning-based solution to address hierarchical resource partitioning on co-scheduling on modern GPUs.
5. Finally, we illustrate that our approach is successful in concurrent establishment of resource partitioning and co-scheduling group selections, while achieving a throughput improvement of up to 87.3% compared to the time-shared scheduling.
6. Additionally, our evaluations also cover comparison with prior studies [4, 71].

2 Background and Related Work

This chapter provides additional details on the background information necessary for a comprehensive understanding of the concepts referenced throughout this work. We provide foundational information related to Profiling Tools, Co-scheduling, Resource Management Features, and Reinforcement Learning.

2.1 Profiling Tools

2.1.1 CPU

Linux Perf

In this section, we will describe the Linux Perf tool, also known as `perf_events` [63]. Perf is a versatile tool capable of instrumenting CPU performance counters, tracepoints [85], kprobes [36], and uprobes [84]. It excels in lightweight profiling capabilities. Perf started as a tool for leveraging the performance counters subsystem and has evolved with numerous improvements to encompass tracing capabilities [63].

CPU performance counters serve as the foundation for profiling applications using `perf`. These counters are specialized hardware registers that count crucial hardware events, including but not limited to instructions executed, cache-misses, and branches misses [63].

Perf tool provides various commands for collecting and analyzing performance data, including but not limited to: `perf stat`, `perf record`, and `perf report` [83]. For our work, only `perf stat` is relevant; thus, we will delve further into its details. For supported events, `perf stat` maintains a running count during process execution [83]. Occurrences of events are subsequently aggregated and printed at the end of execution [83]. A sample output of `perf stat` command has been shown in Listing 2.1 with few selected performance counters.

```
$ perf stat -e duration_time,task-clock,context-switches,cpu-cycles,
   instructions ls

   Performance counter stats for 'ls':
   2,039,833 ns duration_time # 1.494 G/sec
```

```
       1.37 msec task-clock # 0.669 CPUs utilized
       2 context-switches # 1.465 K/sec
3,361,002 cpu-cycles # 2.462 GHz
3,249,994 instructions # 0.97 insn per cycle
0.002039833 seconds time elapsed
0.000000000 seconds user
0.001917000 seconds sys
```

Listing 2.1: Usage example of `perf stat` command

2.1.2 GPU

NVIDIA Nsight Compute Framework

NVIDIA Nsight Compute Framework offers an interactive profiler tailored for CUDA and NVIDIA OptiX, providing performance metrics and API debugging capabilities via both a user interface and a command-line tool [56]. In this work, we will utilize Nsight Compute for profiling the CUDA kernels. NVIDIA Nsight Compute CLI (`ncu`) provides a non-interactive way to profile applications from the command line [53]. It can print the results directly on the command line or store them in a report file. In the execution shown in Listing 2.2 for the `ncu` command, we can specify the profiling report name with the `-o` option.

```
$ ncu -o profile CuVectorAddMulti.exe

    [Vector addition of 1144477 elements]
    ==PROF== Connected to process 5268
    Copy input data from the host memory to the CUDA device
    CUDA kernel launch A with 4471 blocks of 256 threads
    ==PROF== Profiling "vectorAdd_A" - 0: 0%....50%....100% - 46 passes
    CUDA kernel launch B with 4471 blocks of 256 threads
    ==PROF== Profiling "vectorAdd_B" - 1: 0%....50%....100% - 46 passes
    Copy output data from the CUDA device to the host memory
    Done
    ==PROF== Disconnected from process 5268
    ==PROF== Report: profile.ncu-rep
```

Listing 2.2: Usage example of `ncu` command [53]

2.2 Co-scheduling

As compute nodes in HPC systems become more fatter and richer, it is getting more challenging to fully utilize all node resources by a single application. Memory-intensive applications typically require only a fraction of in-node compute resources, while compute-intensive applications may not completely utilize the abundant bandwidth resources the node offers. Several factors contribute to this trend: firstly, not all programs exhibit sufficient parallelism to leverage available compute resources for significant speedup, as governed by Amdahl's law [3]. Secondly, memory-intensive applications face limitations in speedup due to constrained memory bandwidth, rendering increased compute resources ineffective – a phenomenon known as the memory-wall problem [26]. Also, in GPUs, the compute resources are also becoming heterogeneous with different types of units (e.g., matrix engines, regular FP64 units, integer units, etc.), and depending on their usages, power can also be under utilized and wasted [4].

A promising solution to address resource waste is through *co-scheduling*, where multiple applications or jobs are concurrently executed on the same node in a space-sharing manner, which has been widely studied for servers and HPC systems [8, 73, 14, 15, 97]. By co-locating different types of applications that require complementary resources, the resource wastes can be significantly reduced.

Figure 2.1 presents an example of co-scheduling for two jobs at a time. In the given example, resources (cores/LLC/bandwidth) are partitioned for co-location of multiple jobs at a time, hence it is also referred as *space shared scheduling*. Note that the

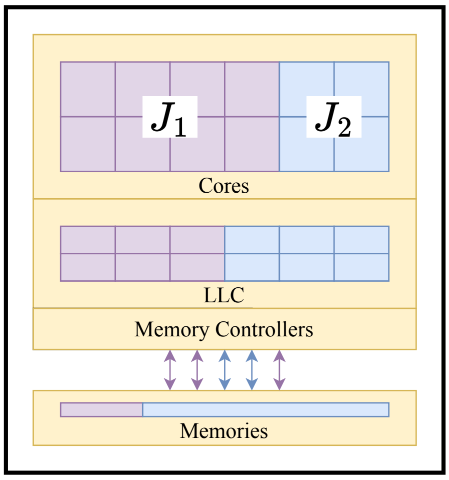


Figure 2.1: An example illustrating co-scheduling for two jobs (J_1 and J_2)

number of jobs co-located at any given time determines the concurrency level of the system. To fully leverage the advantages of co-scheduling, it is important to carefully select jobs for co-scheduling, hence we explicitly target job selections in our work. The major drawback of co-scheduling is that it induces interference effects among co-running applications due to the contentions on shared resources (e.g., shared caches and memory controllers). Therefore, resource partitioning is another important aspect which we discuss in the following section.

2.3 Resource Management Features

In this section, we explore the range of resource management features available for modern HPC systems. Subsequently, in the following sub-sections, we separately describe these features for CPUs and GPUs, respectively.

2.3.1 CPU

For CPU, we will examine three features related to the allocation and partitioning of compute, cache, and memory bandwidth. Initially, with focus on NUMA systems, we will explore the `numactl` command, which assists in specifying core and memory mappings on such systems. Nextly, we will look more closely into the Intel's Resource Director Technology which provides the required framework for partitioning cache and memory bandwidth.

Numactl

Numactl provides support for NUMA policies on Linux. `numactl` executes processes with a designated NUMA scheduling or memory placement policy [43]. Furthermore, numactl has the capability to establish a persistent policy for shared memory segments or files. The following mappings can be done using the `numactl` command.

1. **Core Mapping:** `numactl` can be used to bind physical cores to the application. This can be done with the help of `-physcpubind=cpus` or `-C cpus` option. The parameter `cpus` can be used to pass the cpu numbers. This parameter accepts "`all`", comma-separated cpu numbers and range of cpus as valid inputs. With such core binding, the application can only utilize the specified physical cores.

2. **Memory Mapping:** `numactl` also helps to set the memory placement policies. There are multiple options available, including:

   (1) *first-touch*: It is the default memory placement policy. The memory mapping happens to the NUMA node that first uses it.

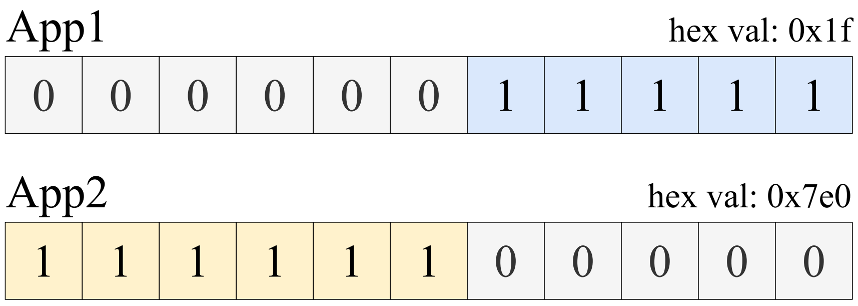


Figure 2.2: Illustration demonstrating the utilization of CBM for CAT: the last 5 cache ways are allocated to App1, while the remaining are assigned to App2. The hexadecimal value corresponding to the binary bitmask is indicated in the top-right corner.

(2) *local-alloc*: This option is useful when memory placement shall always be performed on the current NUMA node where the program is being executed. This policy can be set with `-localalloc` or `-l` option.

(3) *round-robin*: Memory will be allocated using round robin on nodes. When memory cannot be allocated on the current interleave target fall back to other nodes. Round robin allocation can be done by setting memory interleave policy by using `-interleave=nodes` or `-i nodes` option, where `nodes` sets all the NUMA nodes to be considered for interleaving. Here, setting `nodes` value to "`all`" shall use all NUMA nodes.

Intel's Cache Allocation Technology (CAT)

Intel's Cache Allocation Technology (CAT) is part of the Intel's Resource Director Technology which empowers the Operating System (OS) to specify the amount of cache space dedicated to an application [31]. This functionality facilitates the allocation of cache resources based on the application priority or Class of Service (COS). For CAT, the COS definitions are established using a Capacity Bitmask (CBM), defining the relative amount of cache space available to the application. The CBM is provided as a parameter to the `rdtset` command, configuring the cache ways accessible to the application. For example, the system illustrated in Figure 2.2 features 11 cache ways. In this configuration, we can partition the last-level cache for `App1` and `App2`, implementing the required allocation through the command shown in Listing 2.3. Note that `-t` option is used to pass the hexadecimal value of the CBM along with the associated cores, whereas `-c` and `-k` options are used for passing the cores and job command respectively.

```
$ rdtset -t 'l3=0x1f;cpu=0-5' -c 0-5 -k ./App1 &
$ rdtset -t 'l3=0x7e0;cpu=6-11' -c 6-11 -k ./App2 &
```

Listing 2.3: Usage of `rdtset` command for partitioning Last-Level Cache.

Intel's Memory Bandwidth Allocation (MBA)

Intel's Memory Bandwidth Allocation (MBA) is another feature from the Intel's Resource Director Technology that provides indirect and approximate control over the memory bandwidth available to an application. This feature offers to regulate applications that might excessively use bandwidth relative to their priority [31]. Therefore, it helps in partitioning memory bandwidth according to the specific requirements of each application. The MBA works by defining the throttling value or an approximate maximum bandwidth cap. This throttling value can be defined in terms of the percentage of the total memory bandwidth. As an example in Listing 2.4, `App1` and `App2` are allocated 30% and 70% of the memory bandwidth respectively.

```
$ rdtset -t 'm=30;cpu=0-5' -c 0-5 -k ./App1 &
$ rdtset -t 'm=70;cpu=6-11' -c 6-11 -k ./App2 &
```

Listing 2.4: Usage of `rdtset` command for partitioning Memory Bandwidth.

2.3.2 GPU

In case of GPUs, we will focus on the partitioning features available on the modern NVIDIA GPUs. We explain two features in this section: (i) a finer-grained logical partitioning using NVIDIA Multi-Process Service (MPS), (ii) and a coarse-grained physical partitioning using NVIDIA Multi-Instance GPU feature (MIG).

NVIDIA's Multi-Process Service (MPS)

The Multi-Process Service (MPS) is used to allow logical sharing of computational resources among multiple programs. It is a software-based mechanism that can be used to assign process to Streaming Multi-processor (SM) with arbitrarily defined rates. The MPS is implemented as a client-server runtime of the CUDA API, comprising three key components [52]:

1. A control daemon responsible for initiating/stopping the server and coordinating client-server connections.
2. A client runtime, accessible to any CUDA application, facilitating GPU execution.

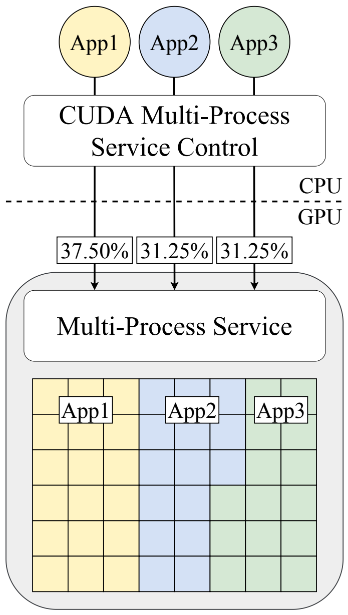


Figure 2.3: Overview of the Multi-Process Service (MPS) (*Re-drawn Version, Original Source:* [52])

3. A server process that promotes concurrency by enabling shared connections to the GPU for clients.

Figure 2.3 illustrates the overview of the working of MPS. In this figure, three applications (App1, App2, App3) are concurrently executed on the GPU using MPS. Notably, MPS allows the allocation percentage of Streaming Multi-processors (SMs) to each application to be set arbitrarily, enabling a finer-grained resource partitioning.

To launch the MPS service on a GPU, we need to execute `nvidia-cuda-mps-control -d` command which starts the control daemon. Note that before the launch of the control daemon, it might be useful to set relevant environment variables such as `CUDA_MPS_PIPE_DIRECTORY` and `CUDA_MPS_LOG_DIRECTORY`. For controlling the allocation percentage for an application, `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` variable can be used at the time of application launch. Listing 2.5 shows the launch of applications corresponding to the Figure 2.3.

```
$ CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=37.50 ./App1 &
$ CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=31.25 ./App2 &
$ CUDA_MPS_ACTIVE_THREAD_PERCENTAGE=31.25 ./App3 &
```

Listing 2.5: Launching multiple applications concurrently using NVIDIA MPS.

As we can see, MPS provides a flexible resource allocation scheme for computational resources. However, it does not offer the ability to partition shared resources like Last-Level Cache (LLC) and High Bandwidth Memory (HBM). Therefore, using MPS may lead to increased contention for shared resources among concurrent applications. Hence, it is recommended to use cooperating workloads with MPS.

NVIDIA's Multi-Instance GPU (MIG)

Multi-Instance GPU is a recent feature introduced in the latest NVIDIA GPUs from the Ampere generation [55]. This new feature enables GPUs to be securely partitioned into up to seven distinct GPU Instances for CUDA applications, ensuring multiple users have separate GPU resources for optimal GPU utilization [55]. This feature proves especially beneficial for workloads that do not fully saturate the GPU's capacity, and in such cases concurrent execution can lead to optimal GPU utilization.

We will discuss further about the NVIDIA A100 GPU Architecture. As shown in Figure 2.4, a single A100 GPU comprises of multiple GPCs (Graphics Processing Clusters), and each GPC is made up of multiple Streaming Multi-Processors. A single SM has its own private resources including local instruction/data cache, a warp scheduler, a dispatcher, a register file, and other functional units. Whereas other resources such as Last-Level Cache (LLC) and High Bandwidth Memory (HBM) are shared by GPCs.

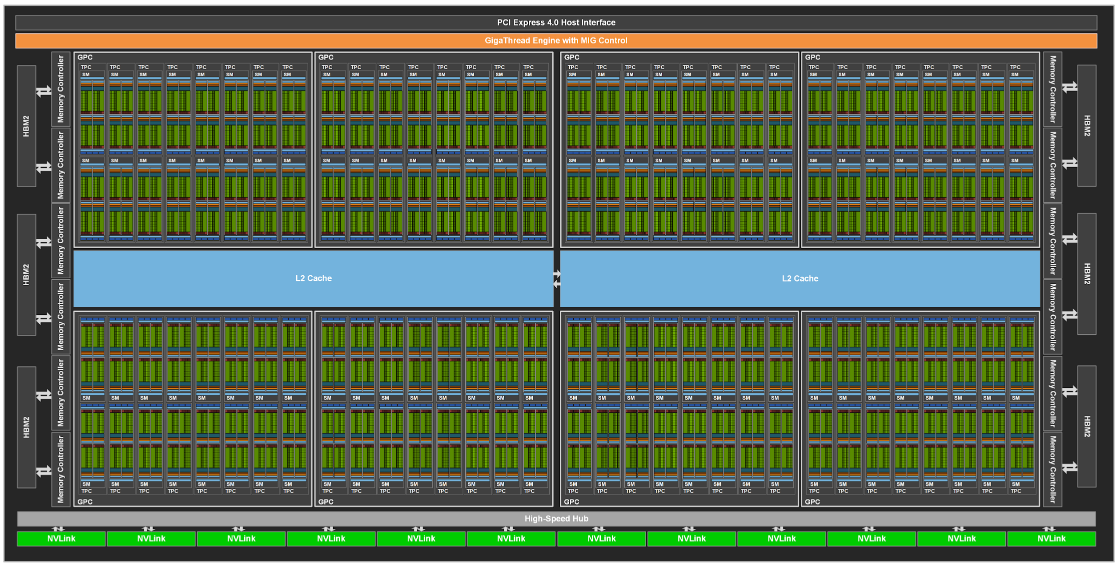

PCI Express 4.0 Host Interface
GigaThread Engine with MIG Control
GPC
TPC
SM
L2 Cache
Memory Controller
HBM2
High-Speed Hub
NVLink

Figure 2.4: A100 GPU Chip Architecture (*Source:* [57])

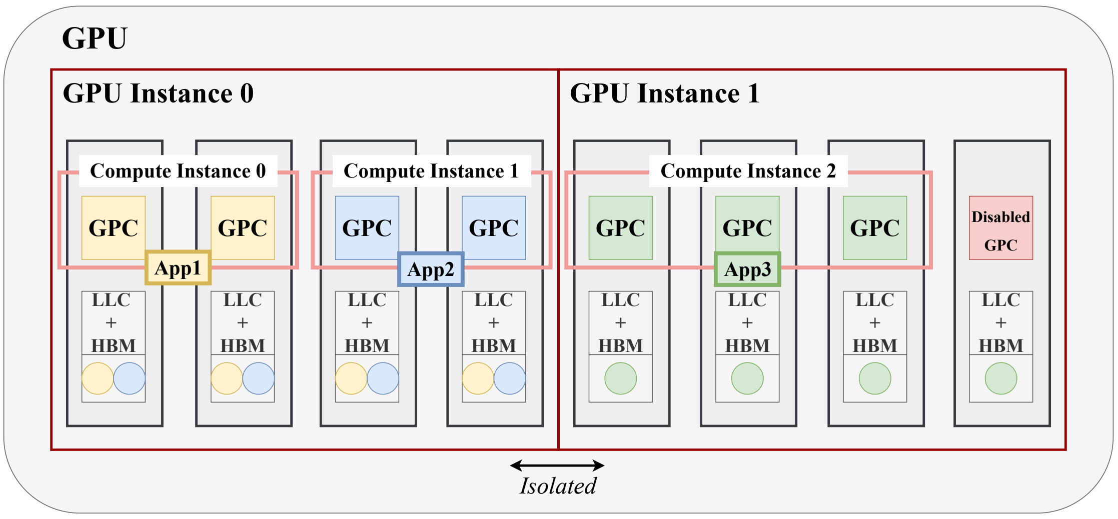


Figure 2.5: Overview of the Multi-Instance GPU (MIG) Feature

MIG enables the resource partitioning at the granularity of GPCs in a hierarchial manner, as described below:

1. Initially, it is used to partition GPU into one or more GPU Instances (GI), ensuring complete isolation between these instances — no shared resources exist between them.

2. Subsequently, these GIs need to be configured with one or more Compute Instances (CI), which share memory resources within the GI but exclusively utilize compute resources at the granularity of GPC.

Note that for performing such setup, we first need to enable the MIG feature using the command: `nvidia-smi -mig 1`. Figure 2.5 illustrates the coarse-grained physical partitioning using MIG. Note that when enabling the MIG feature, 1 GPC is disabled. In this example, out of 7 available GPCs, 2 GIs are created with 4 and 3 GPCs respectively. Additionally, 2 and 1 CIs are created on each GI respectively. With this setup, `App1` and `App2` share LLC/HBM, while `App3` runs with complete isolation. The MIG setup, as described in Figure 2.5, can be configured using the `nvidia-smi` command. Table 2.1 lists the available GI profiles on the A100 GPU. For example, profile `MIG 4g.20gb` and `MIG 3g.20gb` have been used for creating GPU instances illustrated in the Figure 2.5. Additionally, corresponding CI profiles can be chosen for each GPU instance. Note that a MIG device is fully configured and ready to use only after creating Compute Instances (CIs).

Table 2.1: GI Profiles available on A100 GPU

Name	SM Count	Memory (GiB)
MIG 1g.5gb	14	4.75
MIG 1g.5gb+me	14	4.75
MIG 1g.10gb	14	9.62
MIG 2g.10gb	28	9.62
MIG 3g.20gb	42	19.50
MIG 4g.20gb	56	19.50
MIG 7g.40gb	98	39.25

It is noteworthy that MIG partitioning operates at the granularity of a GPC, lacking the flexibility seen in MPS. Therefore, MIG is well-suited for coarse-grained partitioning. Furthermore, in contrast to MPS, the creation of distinct GIs provides capabilities for shared resource isolation. However, disabling one GPC results in a reduction of compute resources.

2.4 Reinforcement Learning

Reinforcement Learning is a machine learning paradigm in which an agent takes actions, learning through interactions with the environment to maximize cumulative rewards [78]. The objective is to empower an agent to explore its environment through interactions, collect rewards, engage in trial-and-error processes, and eventually generalize its learning to execute an optimal set of actions that maximizes the overall reward. It is well-suited for problems involving sequential decisions. Resource management decisions, in our case, can be formulated as a sequence of actions taken to create and execute a scheduling policy, making reinforcement learning particularly suitable for our scenario.

2.4.1 Markov Decision Process

Markov Decision Process (MDP) provides the mathematical framework for modeling decision making in situations where the cost/reward and transition functions rely solely on the current state of the system and the current action [64]. MDP can be mathematically defined by the tuple (S, A, P, R), where: (i) S is a set of states, (ii) A is a set of actions, (iii) P is the state transition probability matrix, and (v) $Rw : S \times A \rightarrow \mathbb{R}$ is a reward function. A policy function $\pi : S \times A \rightarrow [0, 1]$ gives the probability of taking action a when in state s. The main objective in MDP is to find a policy π such that it maximizes the expected value of the cumulative rewards. Typically, a discount factor $\gamma \in [0, 1]$ is also used to provide appropriate weight to the rewards. The objective function can be expressed as following:

$$max \quad \mathbb{E}[\sum_{t=0}^{T} \gamma^t Rw^{a_t}(s_t, s_{t+1})] \tag{2.1}$$

where we need to maximize the cumulative reward for making series actions from timestep 0 to T. Here, as γ tends to zero, the motivation is to maximize immediate rewards rather than focusing on long-term rewards. Note that the presented mathematical framework serves as a foundation for developing additional concepts for reinforcement learning.

2.4.2 Overview

Using the described framework in Section 2.4.1, we can illustrate reinforcement learning as a series of interactions of an agent with an environment, as shown in Figure 2.6. At a timestep t, the overall system can be represented by the state s_t. The agent takes an action a_t, receives a reward rw_t, and undergoes a transition to state s_{t+1}. With this setup, agent shall continue to interact with the environment until it maps the possible state-action pairs to corresponding rewards, and eventually learn to make the optimal set of actions for maximizing the cumulative reward. Further details about these elements are described as follows.

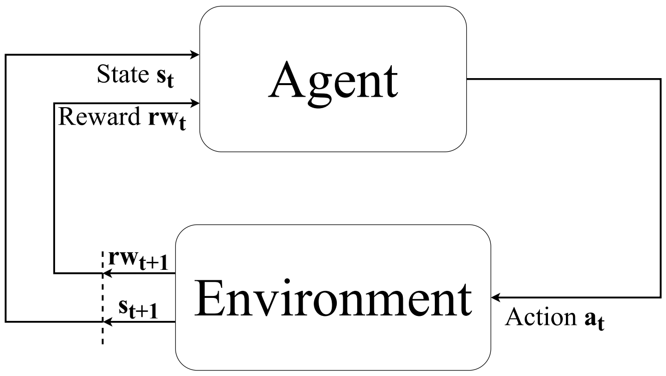


Figure 2.6: Overview of Reinforcement Learning (*Re-drawn Version, Original Source:* [69])

Environment

An environment is the world in which an agent learns through interactions. The environment must be capable of deducing the overall system state and providing the correct representation to the agent. Additionally, the environment must have a reward function that guides the agent in its learning process.

Agent

An agent is an actor and learner in reinforcement learning responsible for making decisions based on the given state of the system. The role of the agent is to gather information about the environment through reward signals and accurately map state-action pairs to corresponding rewards. Through repetitive interactions and trial-and-error, the agent learns to take optimal actions to reach the goal state.

State

In the RL framework, the representation of the current situation of the system is defined as the state. The state should encompass all the relevant information necessary for making decisions about actions. At timestep $t = 0$, the environment provides the initial state of the system, and the objective is to reach the *goal state* through a series of actions. The set of all possible states of the system defines the state space S.

Action

An action is a decision made by the agent based on the current state of the system, causing a transition from the current state to the next state. The set of all possible actions defines the action space A. Depending on the scenario, action space can be either *discrete* or *continuous*. In a discrete action space, a finite set of actions are available to the agent, while in a continuous action space, any real number from a range of values can be used.

Reward

A reward signal defines the goal of the RL problem. For every action, the environment sends a reward signal as a numerical value to the agent. As the agent's goal is to maximize the cumulative reward, the reward signal determines what is a good or bad action at a given state of the system.

2.4.3 Optimization Methods

In the given formulation in Section 2.4.1, we need to optimize for the cost function in a such way that it maximizes the overall reward. For solving the mentioned problem with reinforcement learning, two main classes of methodologies are available:

1. **Model-based RL:** Involves constructing a model of the environment through the analysis of state transitions and outcomes, ultimately generating a functional representation of the environment. In these methods, the emphasis is on estimating the probability distribution and reward function associated with the MDP (2.4.1), and *planning* the agent's behavior based on this acquired knowledge. While model-based methods are more sample-efficient, especially in relatively simple environments, it's important to note that building an accurate model of the environment can be challenging and computationally expensive, particularly in complex scenarios.

2. **Model-free RL:** In scenarios where the dynamics of the environment are complex and cannot be pre-determined, the agent interacts with the environment, observes outcomes via trial-and-error, and *learn* the value associated with sequences of actions over time. In these methods, the emphasis is on estimating the optimal policy or value function without explicitly understanding the underlying dynamics. The flexibility of these methods makes them more applicable to complex and unknown environments.

Due to their simplicity and flexibility, we will be focusing on the model-free reinforcement learning methods. As mentioned above, with model-free learning, the focus is on estimating the policy or value function. These functions are defined as follows.

- **Policy Function:** A policy function defines the agent's behavior in an environment by mapping states to actions and hence represents the agent's decision making. Following on the definition from Section 2.4.1, this function represents the probability of selecting action a when the agent is in state s.

$$\pi : S \times A \to [0,1] \tag{2.2}$$

- **Value Function:** A value function provides an estimate of the expected cumulative reward an agent can receive from a particular state-action pair. It is useful to compare and evaluate the value or goodness of distinct state-action pairs in terms of long-term rewards. Further, there are two main kinds of value functions:
  - *State Value Function (V-function)*: represents the expected cumulative reward when starting from a particular state $s_0 = s$ and following a specific policy

π.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[\sum_{t=0}^{T} \gamma^t Rw_{t+1} | s_0 = s] \tag{2.3}$$

– *Action Value Function (Q-function)*: represents the expected cumulative reward when starting from state $s_0 = s$, taking action $a_0 = a$, and then following a specific policy π.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[\sum_{t=0}^{T} \gamma^t Rw_{t+1} | s_0 = s, a_0 = a] \tag{2.4}$$

Using these mathematical foundations, we will look closer into two state-of-the-art reinforcement learning methods, namely: (i) Q-Learning and (ii) Soft Actor-Critic Method. These methods are *off-policy, model-free* reinforcement learning approaches. Using off-policy approaches encourages the agent to learn from experiences generated by any policy, not necessarily the one currently maximizing rewards. This is important because it enables the agent to explore and learn from sub-optimal moves, contributing to a better understanding of the environment.

Q-Learning

Q-learning is an *off-policy, model-free* reinforcement learning method, where it learns based on the value of an action in a particular state. For any finite MDP, Q-learning tends to find an optimal policy such that it maximizes the expected value of the cumulative reward. Specifically, the main objective here is to map the state-action pairs to the corresponding rewards, and learn to maximize the *Q-function* or the action value function. The optimal Q-function has been defined using the Bellman Optimality Equation [90].

$$Q^{*}(s,a) = \mathbb{E}[Rw_s^a + \gamma \sum_{s' \in S} maxQ^{*}(s',a')] \tag{2.5}$$

In Equation 2.5, the optimal value of taking action a in the state s has been defined. To this end, the reward Rw_s^a refers to the immediate reward of taking action a at state s. Additionally, long-term rewards are maximized for the subsequent series of states and actions. This emphasizes the agent's goal of not only considering immediate rewards but also optimizing its actions for sustained benefit over time. The factor γ (discount factor) is used to apply appropriate weight to the long-term rewards, where if $\gamma = 0$, the agent will be completely myopic and only learn about actions that produce an immediate reward, while a factor approaching 1 will make the agent strive for a long-term high reward.

For learning the optimal Q-function ($Q^*(s,a)$), series of iterations need to performed for updating the Q-values such that the cumulative rewards are maximized. During Q-learning, the Q-values are updated using the rule given in Equation 2.6.

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(rw_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.6}$$

In the Equation 2.6, the paramter α defines the learning rate of the agent. The Q-values are updated iteratively, allowing the agent to learn and estimate Q-values for all possible state-action pairs. Traditionally, a Q-table is employed to store these Q-values, associating them with respective state-action pairs and their corresponding rewards. However, while effective for simpler scenarios, the Q-table approach becomes impractical as the number of states and actions increases, necessitating more sophisticated methods for handling larger state and action spaces. Hence, combining Q-learning with function approximation becomes crucial when dealing with problems involving a large number of states and actions. In this context, Mnih et al.'s work [48] introduced the use of artificial neural networks to approximate the Q-function, enabling the application of the algorithm to more complex scenarios.

Exploration v/s Exploitation: While implementing Q-learning, it is crucial for the agent to explore the environment sufficiently before converging towards optimal policies. The ϵ-greedy approach proves effective in achieving this exploration. In this approach, a parameter ϵ and an ϵ-decay rate are defined. The parameter ϵ determines the probability of the agent taking random actions instead of focusing solely on actions that currently maximize rewards. The initial value of ϵ is set to 1, and with each iteration, it is decreased by the ϵ-decay rate. It is advisable to eventually stabilize the value of ϵ at the minimum value ϵ_{min}, allowing the agent to irregularly take random actions while exploiting the learned optimal policies.

Note that Q-learning is applicable only to discrete action spaces. It is most effective in scenarios where the number of discrete actions is relatively small.

Soft Actor-Critic Method

Soft Actor-Critic (SAC) is another *off-policy*, *model-free* reinforcement learning method that aims to maximize cumulative reward and policy entropy [28]. Entropy characterizes the level of unpredictability in the choice of actions made by the policy. Building on the other model-free methods, SAC has been designed to efficiently handle complex environments. The algorithm works by optimizing the stochastic policy in an off-policy manner and maintains the value function for estimating the state-action values. Standard RL methods aim to maximize for the expected sum of rewards, whereas SAC adapts a more general maximum entropy objective which favors stochastic policies by augmenting the objective function with the expected entropy of the policy [28]. The objective function for SAC can be referred from the Equation 2.7.

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)}[\gamma^t(rw_t + \beta\mathcal{H}(\pi(.|s_t)))] \tag{2.7}$$

In the Equation 2.7, the objective function for policy π aims to maximize the expected values for the discounted rewards rw_t (with dicount factor γ) and the policy entropy $\mathcal{H}$. The temperature paramter β defines the relative importance of the entropy term in the objective function, thus control the stochasticity of the optimal policy. Note that by setting $\beta = 0$, we can recover the objective function for conventional reinforcement learning. The addition of the entropy term encourages exploration by discouraging overly deterministic policies. The parameter β can be used to control the trade-off between maximizing rewards and maximizing entropy.

SAC algorithm involves optimizing several functions during the learning process. Firstly, we define the objective for Q-function, which is parameterized by the function parameters ϕ. Note that during implementation, two separate Q-functions are maintained for improving training stability. In the Equation 2.8, the objective function for the Q-function optimization has been defined using the minimization of the mean squared Bellman error. Note that the used functions are consistent with the previously discussed notations and definitions.

$$\begin{gathered} J(\phi) = \mathbb{E}_{(s_t,a_t)}[\frac{1}{2}(Q_\phi(s_t,a_t) - \hat{Q}(s_t,a_t))^2] \\ \text{where, } \hat{Q}(s_t,a_t) = rw_t + \gamma\mathbb{E}[V(s_{t+1})] \end{gathered} \tag{2.8}$$

Next, the policy π is optimized to maximize the expected cumulative reward and entropy regularization. The policy is parameterized by the function parameters θ. The Equation 2.9 shows the objective function for policy optimization.

$$J_\pi(\theta) = \mathbb{E}_{(s_t,a_t)}[\beta \log(\pi_\theta(a_t|s_t)) - Q_\phi(s_t,a_t)] \tag{2.9}$$

Additionally, SAC optimize for the temperature parameter β as well. The temperature parameter is optimized to encourage the policy to be more or less stochastic. The Equation 2.10 shows the objective function for temperature parameter optimization.

$$J(\beta) = \mathbb{E}_{s_t}[-\beta \log(\pi_\theta(a_t|s_t))] \tag{2.10}$$

Considering this formulation of SAC, the "actor" refers to the policy function that determines the agent's decisions and the "critic" refers to the value functions that critics the action suggested by the actor with the estimated values. The inclusion of entropy term in this framework has been indicated with the term "soft" as it is designed to balance exploration and exploitation. Again, the use of artificial neural networks have been proposed for function approximation in SAC [28].

Note that SAC was originally proposed for continuous action spaces by Haarnoja et al. [28]. However, the effectiveness of SAC in terms of training stability and sample efficiency has led to its adaption to the discrete action settings as well [22].

2.4.4 Role of Deep Neural Networks

Artificial Neural Networks, or simply neural networks are a subset of machine learning techniques which mimics structure and functioning of the human brain. These networks consist of interconnected layers of nodes or artificial neurons, where each node connects to others and has an associated weights and biases. The collective activity of these nodes and their connections allows for identification and extraction of patterns from complex datasets. Therefore, neural networks have found applications in diverse fields, including: medical image classification, social network filtering, behavioral data analysis, financial predictions, and energy demand forecasting [92].

A neural network typically comprises an input layer, one or more hidden layers, and an output layer, each layer featuring multiple nodes. In the context of deep learning, a neural network with multiple hidden layers is referred to as a *deep* neural network, as

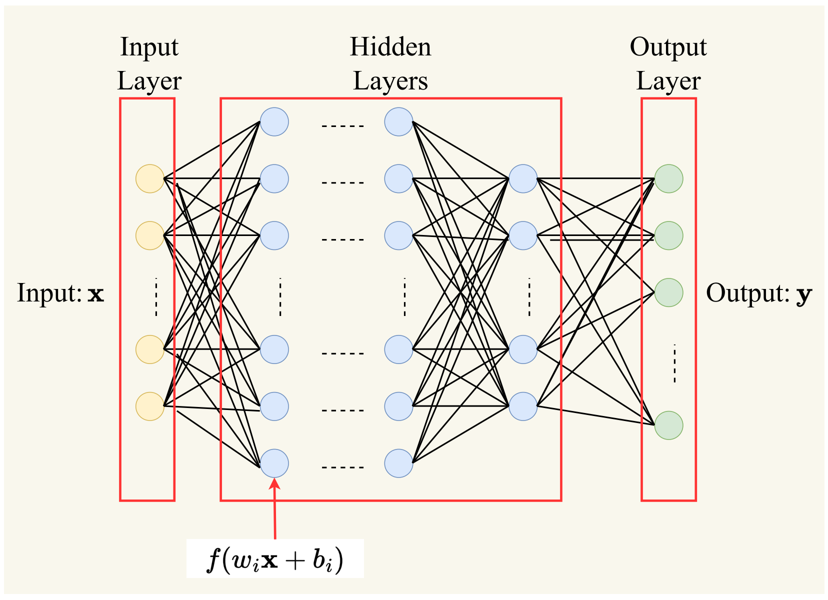


Figure 2.7: Structure of a fully connected deep neural network

depicted in Figure 2.7.

Figure 2.7 illustrates a fully connected deep neural network. In fully connected networks, each neuron in one layer connects to every neuron in subsequent layers. The input vector **x** is fed into the input layer, with the number of neurons in the input layer equal to the input size. Similarly, the output layer's neuron count matches the size of the output vector (**y**). Each node is connected to others with corresponding weights (w_i) and biases (b_i). Neural networks are trained to predict output values for **y**. During training, weights and biases are iteratively adjusted through *backpropagation* [50] to minimize the difference between predicted and actual output values. Further, non-linearity is introduced through activation functions (f), such as `ReLU`, `Sigmoid`, and `Tanh`, enhancing a neural network's capacity to handle complex problems. Following on this formulation, in Figure 2.7, $f(w_i\mathbf{x} + b_i)$ denotes the output of the i^{th} node.

Universal Approximation Theorem suggests that a neural network is capable of learning complex patterns and relationships in data as long as certain conditions are fulfilled, and provided that sufficient data of the corresponding domain is available [81]. Due to inherent capability of deep neural networks to estimate non-linear functions, we utilize deep neural networks as *universal function approximators*, employing reinforcement learning methods that leverage them for approximating functions such as the deep-Q-learning and soft actor-critic method.

2.5 Related Work

2.5.1 NUMA Systems and Optimizations

NUMA-based systems have been a focus of research for several decades, with optimizations in data mapping and process/thread scheduling. The IBM ACE multiprocessor workstation in the 1980s, based on NUMA architecture, prompted studies on various page placement policies [12]. Subsequent work by W. J. Bolosky et al. applied trace-based analyses, indicating that the optimal paging policy relies on architectural parameters [13]. H. Li et al. proposed a locality-based scheduling technique for parallel loops on NUMA [42]. With the dominance of cache-coherent NUMA (CC-NUMA) architecture in the 1990s, B. Verghese et al. introduced an OS-assisted technique to enhance data locality for CC-NUMA [87]. J. Bircsak extended OpenMP to support necessary data placement features for NUMA-based machines [10]. As Dennard scaling ended in the mid-2000s and multi-/many-core architectures took center stage, NUMA effects became prominent even within a node, with multiple sockets enhancing core counts in high-end systems [11, 46]. In response to this architectural shift, studies targeted multi-threaded programs, aiming to minimize thread-to-thread and thread-to-memory overheads for in-node NUMA optimizations [30, 40]. Recent research explored the correlation between thread/data allocations and prefetcher configurations for NUMA systems [70]. *Our work aligns with this literature, integrating NUMA-aware thread/data optimization with emerging shared resource control features, providing a holistic solution that incorporates co-scheduling job set selections using reinforcement learning.*

2.5.2 Co-scheduling and Resource Partitioning on CPUs

Ever since the introduction of multi-/many-core processors to the market, a range of co-scheduling techniques has been put forward for server machines and high-performance computing (HPC) systems. M. Bhadauria et al. investigated the viability of co-scheduling and introduced a scheduling policy based on a greedy approach [8]. H. Sasaki et al. introduced a resource allocation method based on scalability for concurrently scheduled multi-threaded programs [73]. J. Breitbart et al. developed a resource monitoring utility targeting the co-scheduling of HPC applications [14] and proposed a co-scheduling policy taking into account the memory intensity [15]. Q. Zhu et al. focused on CPU-GPU heterogeneous processors and introduced a co-scheduling approach tailored for such systems [97]. I. Saba et al. orchestrated co-scheduling, resource partitioning, and power budgeting across CPU and GPU [71]. J. Choi et al. introduced an application clustering scheme and a co-scheduling algorithm for contemporary processors [21]. V. S. da Silva et al. devised a processor partitioning and co-scheduling method that co-locates a variable number of programs [74]. D. Álvarez et al. implemented a library to achieve system-wide co-scheduling for task-based applications on HPC systems [2]. P. Zou et al. explored the combination of co-scheduling and power capping for clusters [98]. *However, it is noteworthy that these co-scheduling studies primarily focused on job selections and/or resource partitioning, neglecting the integration of memory resource partitioning and NUMA effects.*

In modern microprocessors, where the last-level caches and underlying memory controllers are typically shared among multiple cores, co-scheduled programs can lead to significant contentions on these shared resources. To address the interference effects, an effective approach involves partitioning or isolating shared caches and main memory bandwidth traffic, while optimizing assignments based on the demands. The concept of cache partitioning, along with its associated microarchitectural design, was first proposed by M. K. Qureshi et al., who quantified its effectiveness through simulations [65]. Subsequently, N. Rafique et al. developed a software/hardware mechanism to control memory bandwidth assignments among co-scheduled applications [66]. Motivated by these pioneering studies, the industry has begun to incorporate cache and bandwidth partitioning features in commercial processors [32]. Recent research has explored the benefits of these partitioning features, proposing various techniques to optimize them [5, 51, 60, 93, 59, 20]. Some studies specifically focused on cache partitioning features [5, 51], while others concentrated on memory bandwidth partitioning [60, 93]. J. Park et al. conducted an evaluation of the combination of these two features [59], and R. Chen et al. applied a machine learning approach to optimize the configurations of these hardware features [20]. *In our work, we extend the optimization to encompass the combination of cache and memory bandwidth partitioning, introducing two novel aspects: (1) NUMA-aware resource assignments and (2) job set selections for co-scheduling from a given job queue. These additions contribute to a more comprehensive and refined optimization strategy in the context of shared resource management.*

2.5.3 Co-scheduling and Resource Partitioning on GPUs

S. Pai et al. initially identified resource wastage within a GPU during the execution of a CUDA kernel and investigated the feasibility of GPU multiprocessing through their elastic kernel implementation [58]. I. Tanasic et al. introduced a microarchitectural mechanism for enabling multiprocessing on GPUs without requiring modifications to the kernel [79]. Subsequently, the MPS feature, inspired by these seminal studies, has been integrated into commercial Nvidia GPUs [52]. Various studies have concentrated on software mechanisms to enhance multiprocessing efficiency on GPUs. T. Allen et al. presented Slate, a framework optimizing the combination of co-located processes and dynamically adjusting their scales [1]. smCompactor, similar to Slate, focuses on maximizing resource utilization [19]. C. Reano et al. proposed a secure co-scheduling mechanism that considers memory footprints when co-scheduling processes in a time-sharing manner [67]. In contrast, other studies have emphasized hardware mechanisms to enhance the efficiency of concurrency-controlling features [24, 6, 37]. With the advent of industry-supported physical resource partitioning, such as MIG [55], a few studies have targeted MIG-based partitioning, proposing optimization mechanisms [41, 4, 71]. *The work closest to ours, [71], addresses co-scheduling decision-making and resource partitioning but lacks management of hierarchical partitioning and is limited to co-locating only two programs.*

2.5.4 System Optimizations with Reinforcement Learning

Reinforcement learning, being a versatile approach for optimizing systems through interaction with the environment, has found extensive applications in computing system optimizations. E. Ipek et al. employed reinforcement learning for memory controllers, dynamically selecting an optimal scheduling policy in real-time [33]. Yoo et al. applied reinforcement learning to QLC SSDs to determine various size/threshold parameters [94]. D. Zhang et al. devised a reinforcement learning-based batch scheduler for HPC systems, automatically configuring the priority function [95]. R.

Chen et al. introduced reinforcement learning to optimize resource partitioning on commodity servers for multi-programmed server workloads [20]. Y. Wang et al. proposed a reinforcement learning-based power management mechanism for multi-core processors [88]. P. Zhang et al. implemented reinforcement learning in an ensemble prefetch controller that dynamically selects the best policy from multiple prefetchers [96]. G. Singh et al. suggested an adaptive and extensible data placement using online reinforcement learning for hybrid storage systems [75]. *While these previous studies show promise or have been adopted in production-level systems, they address different problems/components than our current focus.*

3 Optimization Strategy: Methodology Insights

In this chapter, we will present a formal mathematical definition of the problem and then propose a holistic solution employing reinforcement learning. This chapter adapts a general approach and offers a comprehensive overview of the optimization strategy while proposing a methodology that is applicable to both CPU and GPU.

3.1 Problem Definition

In HPC systems, there are typically multiple users trying to schedule their workloads on the system. In this context, HPC schedulers play a crucial role in managing the queuing system, assessing resource availability, and monitoring system health. Furthermore, utilizing co-scheduling without a systematic approach to HPC scheduling can result in inefficiencies, as multiple jobs run concurrently, potentially leading to interference and suboptimal performance in the system. Therefore, in this section, we begin by formulating the HPC scheduling problem in a concise mathematical manner.

Our focus is on addressing the scenario of an over-crowded system characterized by extended queuing times, where there is a persistent availability of jobs ready to run. This situation is common in HPC centers where resource demand often exceeds the available capacity. For a given job queue $\mathcal{Q}$, we define a job window $\mathcal{W}$ to assess the scheduling strategy for a batch of jobs within $\mathcal{W}$ at a time. Figure 3.1 illustrates the formulation of the optimization problem.

As shown in Figure 3.1, jobs are arranged in the job queue $\mathcal{Q}$, and our attention is on the initial $\mathcal{W}$ jobs $(J_1, J_2, \ldots, J_{\mathcal{W}})$ at any given time. We also introduce a concurrency limit $\mathcal{C}_{max}$, where concurrency $\mathcal{C} \leq \mathcal{C}_{max}$ signifies the number of jobs selected for co-execution. Window Size $\mathcal{W}$ and Concurrency Limit $\mathcal{C}_{max}$ define the scheduling behavior and are referred to as the *Scheduling Attributes*: $(\mathcal{W}, \mathcal{C}_{max})$.

Given the system's constraint of limited *total resources* R_T, the task is to formulate *co-schedule* in form of job-set JS$_i$ in a way that efficiently partitions and allocates the resources R_T as $R_i = (r_1, r_2, \ldots)$. Within these constraints, the optimization problem has two primary objectives: (i) *Co-Scheduling*: selecting a set of jobs JS$_i$ to ensure concurrent runs of complementary jobs, and (ii) *Resource Partitioning*: creating partitions of total

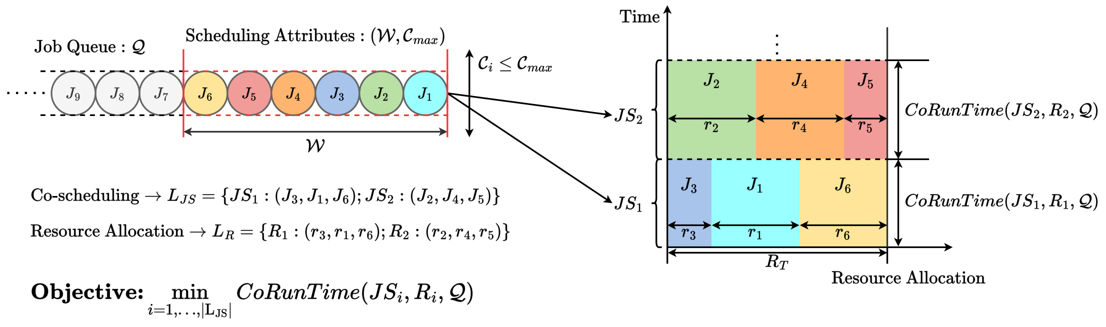

Job Queue : $\mathcal{Q}$
Scheduling Attributes : $(\mathcal{W}, \mathcal{C}_{max})$
$\mathcal{C}_i \leq \mathcal{C}_{max}$
J_9 J_8 J_7 J_6 J_5 J_4 J_3 J_2 J_1
$\mathcal{W}$
Co-scheduling $\rightarrow L_{JS} = \{JS_1 : (J_3, J_1, J_6); JS_2 : (J_2, J_4, J_5)\}$
Resource Allocation $\rightarrow L_R = \{R_1 : (r_3, r_1, r_6); R_2 : (r_2, r_4, r_5)\}$
Objective: $\min_{i=1,\dots,|L_{JS}|} CoRunTime(JS_i, R_i, \mathcal{Q})$
Time
JS_2
JS_1
J_2 J_4 J_5
r_2 r_4 r_5
J_3 J_1 J_6
r_3 r_1 r_6
$CoRunTime(JS_2, R_2, \mathcal{Q})$
$CoRunTime(JS_1, R_1, \mathcal{Q})$
R_T
Resource Allocation

Figure 3.1: Problem Definition

resources into suitable resource chunks, allocating resources as specified by the tuple R_i to the jobs in the co-schedule.

On the right side of Figure 3.1, we illustrate the allocation of system resources over time, with the x-axis representing resource allocation and the y-axis representing execution time. The job sets $\mathrm{JS}_1 : (J_3, J_1, J_6)$ and $\mathrm{JS}_2 : (J_2, J_4, J_5)$ are assigned resources $R_1 : (r_3, r_1, r_6)$ and $R_2 : (r_2, r_4, r_5)$, respectively. For each job set, the objective is to minimize the $CoRunTime$, signifying the overall goal of minimizing the height of the plot. We formulate the optimization problem mathematically as follows.

$$\begin{aligned}
&given && \mathcal{W}, \quad \mathcal{C}_{max}, \quad \mathcal{Q} = \{\mathrm{J}_1, \mathrm{J}_2, ..., \mathrm{J}_{\mathcal{W}}\} \\
&min && \sum_{i=1}^{|\mathrm{L_{JS}}|} CoRunTime(\mathrm{JS_i}, R_i, \mathcal{Q}) \\
&s.t. && CoRunTime(\mathrm{JS_i}, R_i, \mathcal{Q}) \leq SoloRunTime(\mathrm{JS_i}, \mathcal{Q}) && (3.1) \\
& && 1 \leq C_i (= |\mathrm{JS_i}|) \leq C_{max}, \qquad |\mathrm{JS_i}| = |\mathrm{R_i}| && (3.2) \\
& && \forall R_i, \quad r_1 + r_2 + \dots r_{|\mathrm{R_i}|} \leq R_T && (3.3) \\
& && \forall i \in [1, |\mathrm{L_{JS}}|], \qquad |\mathrm{L_{JS}}| = |\mathrm{L_R}| && (3.4) \\
& && \mathrm{JS}_1 \cup ... \cup \mathrm{JS}_{|\mathrm{L_{JS}}|} = \mathcal{Q} && (3.5) \\
& && |\mathrm{JS}_1| + ... + |\mathrm{JS}_{|\mathrm{L_{JS}}|}| = \mathcal{W} && (3.6) \\
&output && \mathrm{L_{JS}} = \{\mathrm{JS}_1, \mathrm{JS}_2, ...\}, \quad \mathrm{L_R} = \{R_1, R_2, ...\}
\end{aligned}$$

During the optimization process within the specified constraints, we derive a list of job sets $\mathrm{L_{JS}}$ alongside a list of corresponding resource partitioning states $\mathrm{L_R}$. The constraint 3.1 represents that co-scheduling the i^{th} set of jobs in $\mathrm{L_{JS}}$ must improve performance compared with time-shared scheduling, i.e., running the jobs one by

Table 3.1: Parameter/Function Definitions

Parameter/Function	Definition
$\mathcal{Q}$	Queuing jobs within the window: $\mathcal{Q} = \{J_1, J_2, \cdots, J_{\mathcal{W}}\}$
$\mathcal{W}$	The number of jobs within the window on the queue
$\mathcal{C}_{max}$	The maximum number of concurrently executed jobs
$\mathrm{L_{JS}}$	A list of job sets to be co-scheduled: $\mathrm{L_{JS}} = \{\mathrm{JS_1}, \mathrm{JS_2}, \cdots\}$
$\mathrm{JS_i}$	ith set of jobs in $\mathrm{L_{JS}}$ to be co-scheduled
$\mathrm{L_R}$	A list of resource partitioning/allocation setups associated with the job sets: $\mathrm{L_R} = \{R_1, R_2, \cdots\}$
R_i	The resource partitioning/allocations for $\mathrm{JS_i}$
$\mathcal{C}_i$ $(= \lvert\mathrm{JS_i}\rvert)$	The concurrency of ith co-scheduled job set
$ExecutionTime(J_k, \mathrm{JS}_i)$	The time taken to execute k^{th} job in the job-set JS_i
$CoRunTime(\mathrm{JS_i}, R_i, \mathcal{Q})$	The total execution time when co-locating $\mathrm{JS_i}$ with R_i (Longest running job determines the execution time of the job-set) $\Rightarrow \max_{k=0,\ldots,\lvert\mathrm{JS_i}\rvert}[ExecutionTime(J_k, \mathrm{JS}_i)]$
$SoloRunTime(\mathrm{JS_i}, \mathcal{Q})$	The total time when executing $\mathrm{JS_i}$ with time sharing (Sum of execution time of all jobs in the job-set) $\Rightarrow \sum_{k=0,\ldots,\lvert\mathrm{JS_i}\rvert}[ExecutionTime(J_k, \mathrm{JS}_i)]$
$JobMixTpt(\mathrm{JS_i}, R_i, \mathcal{Q})$	Relative throughput normalized to that of time-sharing scheduling for a particular job mix (Ratio of $SoloRunTime(\mathrm{JS_i}, \mathcal{Q})$ to $CoRunTime(\mathrm{JS_i}, R_i, \mathcal{Q})$) $\Rightarrow \frac{SoloRunTime(\mathrm{JS_i}, \mathcal{Q})}{CoRunTime(\mathrm{JS_i}, R_i, \mathcal{Q})}$
$Throughput(\mathrm{L_{JS}}, \mathrm{L_R}, \mathcal{Q})$	Overall throughput for the entire job queue ($\mathcal{Q}$) (Weighted sum of $JobMixTpt(\mathrm{JS_i}, R_i, \mathcal{Q})$) $\Rightarrow \sum_{i=1}^{\lvert\mathrm{L_{JS}}\rvert} \frac{\mathcal{C}_i}{\mathcal{W}} JobMixTpt(\mathrm{JS_i}, R_i, \mathcal{Q})$

one using the entire resources exclusively. Constraint 3.2 defines two objectives: (i) it restricts co-scheduling concurrency, i.e., the concurrency ($\mathcal{C}_i$) must be less than or equal to the given upper limit ($\mathcal{C}_{max}$), and (ii) the number of selected jobs in job-set JS_i shall be equal to the generated resource partitions in R_i. Constraint 3.3 signifies that for all resource partitioning states R_i, resource allocation is bounded by the total available resources R_T. Next, constraint 3.4 ensures that these constraints hold for any i

($1 \leq i \leq |\mathrm{L}_{\mathrm{JS}}|$). Constraints 3.5 and 3.6 restrict the job set selections, ensuring they are chosen from the queue ($\mathcal{Q}$) in a mutually exclusive and collectively exhaustive manner. Table 3.1 lists the definitions of parameters and functions used above. **Important:** Note that the metric *throughput*, as used in the following chapters, has been defined as $Throughput(\mathrm{L}_{\mathrm{JS}}, \mathrm{L}_{\mathrm{R}}, \mathcal{Q})$, in Table 3.1.

This optimization problem can be viewed as a variant of the widely recognized strip packing problem [45]. In a typical strip packing problem, there are multiple items, usually with rigid rectangular shapes, and a strip is given. The objective is to minimize the height by arranging all the items within the strip. In our scenario, the shapes of the given items can be altered based on the resource assignment setups (R_i), and each item possesses multiple dimensions. Given that the fundamental strip packing problem is acknowledged as NP-hard [45], ours is also a strongly hard problem, given its high degree of freedom in decision-making.

3.2 High-Level Solution Overview

In this section, we present a high-level overview of the proposed methodology. We introduce a comprehensive approach that addresses the optimization challenge through the application of reinforcement learning.

Figure 3.2 offers an overview of our solution's entire system architecture. As depicted, the comprehensive solution comprises three main components: (1) *offline profiling* for job characterization and profile collection; (2) *offline training* to learn the coefficients of our agent; and (3) *online optimization* to employ the trained agent for decision-making.

3.2.1 Offline Profiling

To characterize applications, we use hardware performance counters to capture the runtime characteristics of jobs on the target system, and the specific counters are detailed in chapters 4 and 5. Profiling is a prerequisite for all co-scheduling targets in both the offline and online phases. During the offline phase, solo-run profiles are collected for all benchmark programs before model training. In the online optimization phase, if a queuing job lacks a profile, it is excluded from the co-scheduling targets. This job is executed exclusively, utilizing the entire system resources while collecting the profile, which is then stored in the *Job Profiles Repository*. If an application is rerun on the system, it is included in the co-scheduling target since its profile is available in the repository. A matching function is necessary to associate each job with its corresponding profile, based on submission information such as the binary path and user ID. In this study, we opt for a straightforward approach, using the application binary path plus name as a key and checking for an associated profile in the repository. While our

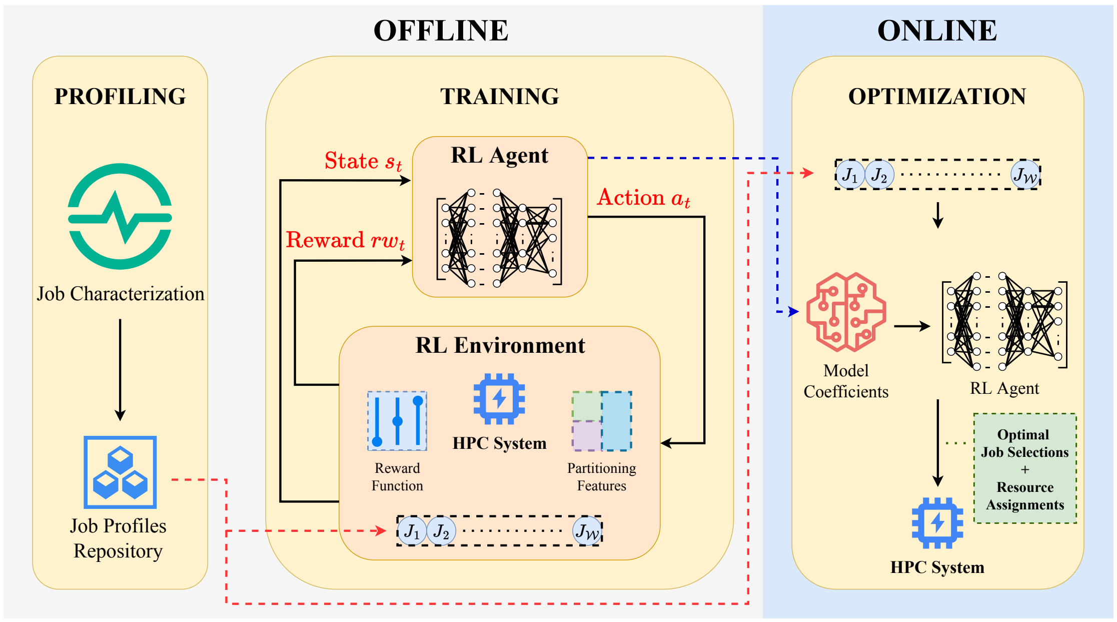


Figure 3.2: Solution Overview

current method is basic, developing a more sophisticated scheme for generating the key from job submission information, considering various factors like input dependency, remains an open challenge. For instance, the characteristics/behavior of an application can depend on its inputs, and there are several promising solutions to compensate for it [17]. In future work, our matching function can be replaced with a more advanced approach.

3.2.2 Offline Training

For the offline model training, we generate variations of benchmark program mixes to be co-located on the target GPU. For each program mix, we systematically explore co-run throughput while adjusting the partitioning setup. This partitioning search relies on reinforcement learning, where we adaptively update the partitioning and resource allocations based on the output of the *reward function*. The reward function considers the co-run throughput. Throughout this process, the *state-action* table, approximated by a neural network in our study, is trained, and the model coefficients in the agent are ultimately determined. It's crucial to note that the model coefficients are hardware-specific and are not transferable to different hardware configurations. However, the training procedure only needs to be conducted once for a specific system. We opt for this offline training approach based on reinforcement learning for several reasons.

Firstly, the job selection and resource partitioning setup may not always be dynamically configurable at runtime. As a result, we cannot adaptively learn optimal configurations for a given set of jobs in the queue ($\mathcal{Q}$) by testing various configurations at runtime. Secondly, during the offline training phase, we employ reinforcement learning instead of using well-known supervised learning with a training dataset. This choice is driven by the impracticality of obtaining a *labeled* dataset. In this context, labeling involves associating a given job mix with the *best* co-scheduling and resource partitioning decisions, necessitating an *exhaustive search* for each job mix (or data) in the dataset.

3.2.3 Online Optimization

In the online phase, we employ an optimization agent to address the optimization problem defined in Section 3.1, utilizing the model generated during the offline phase. The agent treats the optimization as a classification problem, utilizing the model to select sets of co-scheduled job mixes (L_{JS}) and their associated resource allocations (L_R) with the aim of maximizing the system throughput. Notably, in this study, we do not update the model dynamically during the online phase. However, the potential for dynamically refining the trained model stands as a promising avenue for future work.

3.3 Reinforcement Learning Components

In reinforcement learning, an agent learns the optimal action based on the situation to maximize the cumulative reward [78]. As discussed in Section 2.5.4, the objective of this learning approach is to enable the agent to explore the parameter space through interactions with the environment, engage in trial-and-error, and eventually generalize to perform an optimal set of actions to reach the goal state. The various components of reinforcement learning in the context of this work are explained as follows.

1. **Agent:** The agent learns an optimal policy to maximize the accumulation of reward signals during offline training in our approach. In this work, our agent functions as a *co-scheduler* responsible for selecting sets of job mixes and their associated partitioning (L_{JS} and L_R) from the given queue ($\mathcal{Q}$). We set up the agent with deep neural networks for function approximation, enabling it to learn the optimal set of state transitions and maximize the cumulative reward.

2. **Environment:** The environment serves as a black box for the agent. In this work, the environment encompasses the job queue ($\mathcal{Q}$), target HPC system and its hardware features.

$$
\begin{array}{cc|cccccccccccc|}
 & & \multicolumn{4}{c}{\overbrace{\hspace{8em}}^{\text{Job Features}}} & \multicolumn{4}{c}{\overbrace{\hspace{8em}}^{\text{Hardware Configurations}}} & \multicolumn{4}{c}{\overbrace{\hspace{8em}}^{\text{Scheduling Decisions}}} \\
 & & jf_1 & jf_2 & \dots & jf_n & hc_1 & hc_2 & \dots & hc_m & sd_1 & sd_2 & \dots & sd_p \\
\hline
\text{Jobs in Queue}(\mathcal{Q}) & J_1 & jf_1^{J_1} & jf_2^{J_1} & \dots & jf_n^{J_1} & hc_1^{J_1} & hc_2^{J_1} & \dots & hc_m^{J_1} & sd_1^{J_1} & sd_2^{J_1} & \dots & sd_p^{J_1} \\
 & J_2 & jf_1^{J_2} & jf_2^{J_2} & \dots & jf_n^{J_2} & hc_1^{J_2} & hc_2^{J_2} & \dots & hc_m^{J_2} & sd_1^{J_2} & sd_2^{J_2} & \dots & sd_p^{J_2} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & J_{\mathcal{W}} & jf_1^{J_{\mathcal{W}}} & jf_2^{J_{\mathcal{W}}} & \dots & jf_n^{J_{\mathcal{W}}} & hc_1^{J_{\mathcal{W}}} & hc_2^{J_{\mathcal{W}}} & \dots & hc_m^{J_{\mathcal{W}}} & sd_1^{J_{\mathcal{W}}} & sd_2^{J_{\mathcal{W}}} & \dots & sd_p^{J_{\mathcal{W}}} \\
\hline
\end{array}
$$

Figure 3.3: State Representation for Resource Management

3. **State:** The representation of the current system situation is defined as the state. It should contain all relevant information necessary for deciding actions. In our approach, the state of the system includes all jobs in the current job window ($\mathcal{Q} = J_1, J_2, \cdots, J_{\mathcal{W}}$) along with their job features characterized by their profiles. For example, Figure 3.3 illustrates state representation that can be a good candidate during reinforcement learning in our approach. In Figure 3.3, the state representation includes job characteristics and other parameters updated by the agent's decisions, such as hardware configurations (resource allocations) and scheduling decisions. Hence, a state can be represented by a vector $s \in \mathbb{R}^{\mathcal{W} \times (n+m+p)}$, where n, m, and p are the count of values for job features, hardware configurations, and scheduling decisions, respectively.

4. **Action:** Actions in our approach can involve decisions for selecting sets of co-scheduled job mixes and corresponding resource allocations ($\mathrm{L_{JS}}$ and $\mathrm{L_R}$). We limit our search for the optimal set of decisions to a discretized parameter space for resource allocation combinations. Consequently, our action space is also discrete.

5. **Reward:** A reward signal defines the goal of reinforcement learning [78]. For every action, the agent receives the reward signal as a numerical value. As the agent's goal is to maximize the cumulative reward, the reward signal quantifies and evaluates an action at a given state of the system. The details of the setup for this reward function will be provided in chapters 4 and 5. In this work, we define two types of rewards: (1) *Intermediate Reward*: When an agent is building a schedule, it is essentially making different decisions about resource partitioning for concerned jobs. For each of these actions, we define certain rewards based on

heuristics. (2) *Final Reward*: When an agent has successfully created a schedule, it executes the schedule and hence transitions to the final state. In this case, the reward signal can be directly taken from the measured overall speed-up as compared to running the job in the time-shared manner.

4 Harmonized Resource Management on NUMA Systems

In this chapter, our emphasis is on achieving "*Harmonized*" resource management on NUMA systems. As discussed in Section 2.3.1, contemporary NUMA systems offer resource partitioning features like `numactl` and `rdtset`. These features allows for diverse resource assignments on a NUMA system. This chapter will initially introduce these diverse resource assignment policies, followed by empirical observations when employing these policies. Subsequently, we will delve into the implementation details of our reinforcement learning-based approach, specifically designed to harmonize *co-scheduling* and *resource assignments* on NUMA systems. We will then validate our approach through experimental evaluations.

4.1 Overview

We focus on HPC systems that consists of multiple NUMA domains and offer cache and bandwidth partitioning features controllable through software. The resource partitioning for cache and memory bandwidth are explained as follows: (1) *Last-level Cache:* we focus on partitioning last-level cache by dividing and allocating cache to concurrently running jobs at the granularity of the cache ways; (2) *Memory Bandwidth:* we assume that the memory controllers have the capability to partition and prioritize the memory access traffic of concurrently running jobs by defining the memory bandwidth utilization limit. As discussed in Section 2.3.1, these features are commonly available in modern server-class commercial microprocessors [32]. Our objective is to co-schedule multi-threaded or multi-processed jobs/applications and simultaneously determine the resource assignments, including cores, cache ways, and bandwidth.

For instance, Figure 4.1 illustrates the system used in our evaluations. This system comprises of two CPU sockets, both of them supporting the Intel RDT features [32] for allocating cache ways and memory bandwidth to each active job (represented as J_* in the figure). The connections between these NUMA domains are facilitated by an interconnect known as UPI (Ultra Path Interconnect). The allocation of cores and memory is managed through the `numactl` tool set [38], while cache ways and memory

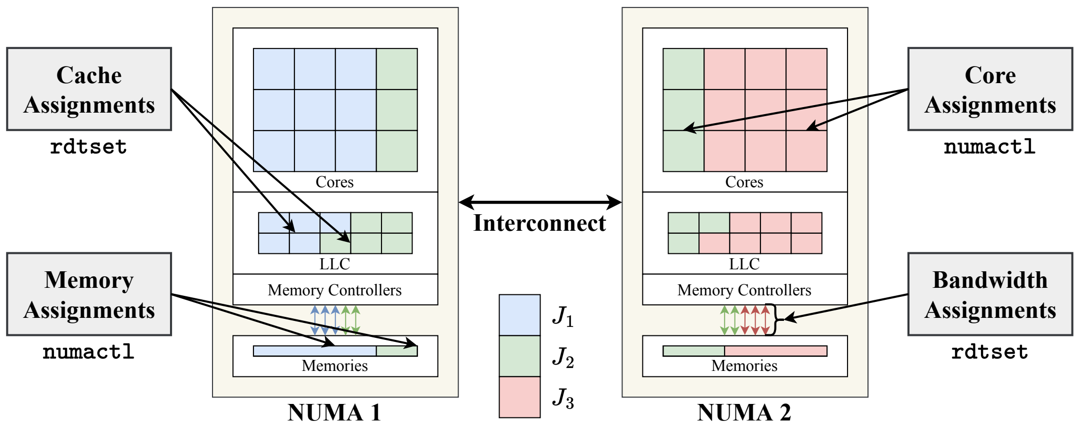


Figure 4.1: Overview of the Target NUMA System

bandwidth assignments are executed using the `rdtset` package [32]. Note that further details about the evaluation setup is provided in Table 4.1.

4.1.1 Allocation Policies

We explore two distinct core mapping policies for NUMA systems, namely *compact* and *distributed*. With compact allocation policy, cores are predominantly allocated from the same NUMA domain, whereas distributed allocation policy involves choosing cores from different NUMA domains in a round-robin fashion. In Fig. 4.1, the distributed policy is used for J_2, whereas the compact policy is applied to the others. In both scenarios, various memory mapping options, including *first touch*[12, 87], *round robin*, and *local alloc*, are considered to optimize memory accesses for the specific co-running jobs and the chosen core mapping.

The compact option is preferable when NUMA interconnect is the performance bottleneck. This case may occur when the running job involves frequent inter-core communications or irregular memory accesses that are sparse and intensive. Conversely, the distributed option is suitable when memory references are more regular or localized within each NUMA domain, potentially providing additional cache capacity and memory bandwidth for the job since these resources are also distributed. After determining the co-run job set and core/memory mappings, cache and bandwidth partitioning features are applied to mitigate interference effects among co-located jobs on each NUMA domain.

4.2 Observations

In this section, we provide observations aimed at understanding the impact of employing different resource assignment policies on a NUMA system.

4.2.1 Throughput v/s Allocation Policy

Figure 4.2 presents a throughput comparison across different core affinity policies for various job pairs. The y-axis enumerates distinct policies for each job pair, while the x-axis represents the relative throughput normalized to that of time-shared scheduling using exclusive solo runs. We evaluate both compact and distributed core affinities with and without employing cache/bandwidth partitioning features. The best policy out of the four is denoted with "*", highlighting the optimal choice for each job mix. When cache/bandwidth partitioning features are enabled, we conduct an exhaustive search to select the best setup, testing all possible configurations and choosing the one that maximizes throughput. This includes optimizing the number of cores assigned to co-scheduled applications and selecting memory mapping from three different options (detailed in Section 4.4) to maximize co-run throughput. The specific search space related to hardware assignments is detailed in Section 4.4.

As depicted in Figure 4.2, the emerging cache/bandwidth partitioning features

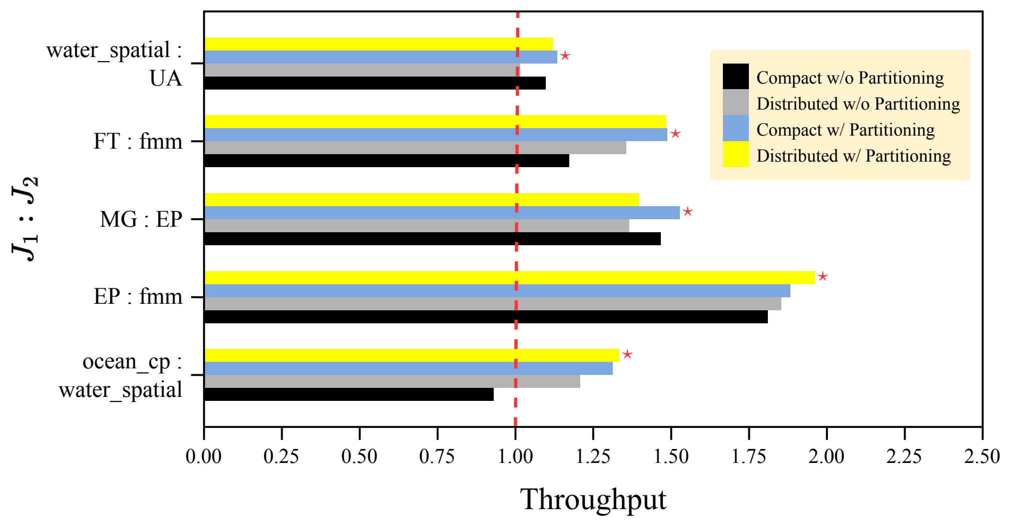


Figure 4.2: NUMA Throughput comparison among different resource assignment policies for various job pairs ($J_1 : J_2$). The ⋆ indicates the best assignment policy for a given job pair.

prove highly effective for specific job mixes, such as `ocean_cp:water_spatial` and `FT:fmm`. This efficiency stems from the memory-intensive or cache-friendly nature of these programs. Isolating/partitioning these memory resources significantly mitigates interference effects, leading to substantial throughput improvement when resource assignments are configured accordingly. However, for other workloads like `MG:EP` and `water_spatial:UA`, these new partitioning features are less effective. The primary reason is that inter-NUMA communications, which can induce interference effects on the interconnect, become a bottleneck for these workloads. Therefore, cache/bandwidth partitioning features do not contribute to throughput improvement, making core affinity setup crucial in such cases. Hence, depending on the selected jobs for co-location, a careful choice between compact and distributed affinity policies is necessary. Simultaneously, selecting an appropriate job pair is also crucial — for example, mixing `water_spatial` with `ocean_cp` outperforms doing so with `UA`, as illustrated in the figure.

4.2.2 Optimal Resource Assignment

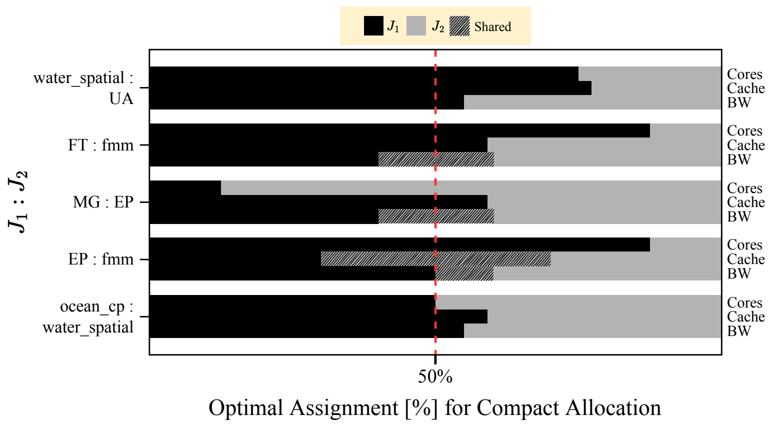


Figure 4.3: Optimal NUMA resource assignment setup for the "Compact w/ Partitioning" allocation policy

In the subsequent figures, Figure 4.3 and Figure 4.4, we present a detailed breakdown of resource assignments for various application pairs optimized under *compact* and *distributed* core affinity options, respectively. The Y-axis enumerates different resources per job mix, while the X-axis aggregates the rates of resource allocation. Additionally, for cache and bandwidth partitioning, we also consider the *shared* option, where resources can be utilized by both co-scheduled programs. As evident in the figures, the choice of core affinity (*compact* or *distributed*) significantly influences decisions regarding resource

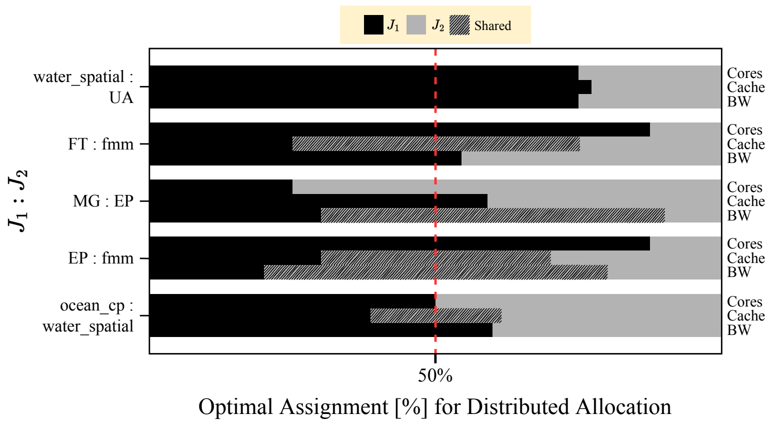


Figure 4.4: Optimal NUMA resource assignment setup for the "Distributed w/ Partitioning" allocation policy

partitioning setups, including cores, cache, and bandwidth partitioning. Simultaneously, the selection of a specific job pair markedly impacts decisions on these resource assignments, as optimal configurations heavily depend on the given job mix.

4.2.3 Throughput v/s Core Allocation

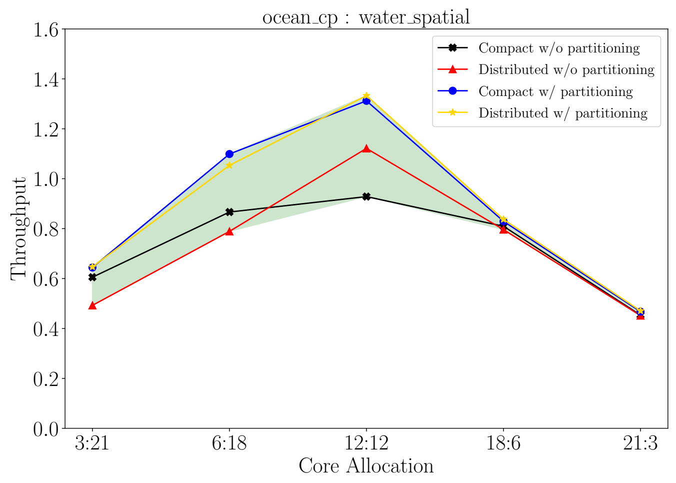


Figure 4.5: Throughput as a function of core allocation for various NUMA allocation policies (`ocean_cp:water_spatial`)

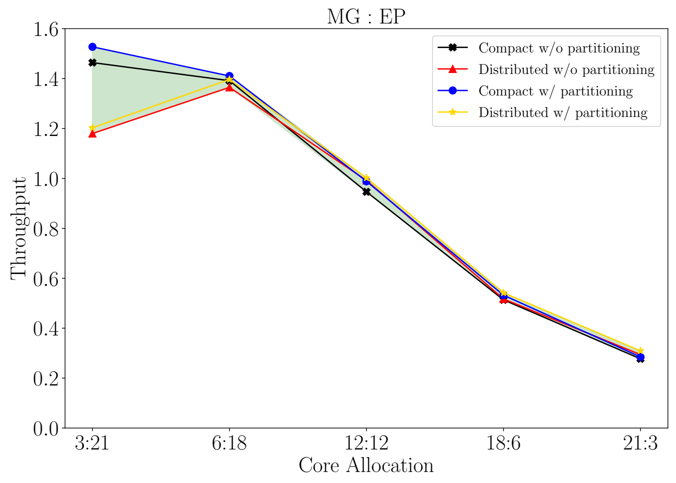


Figure 4.6: Throughput as a function of core allocation for various NUMA allocation policies (`MG:EP`)

Lastly, in Figure 4.5 and Figure 4.6, we illustrate the co-run throughput in relation to the number of cores, considering four different resource assignment options when co-scheduling `ocean_cp` and `water_spatial` or `MG` and `EP`. The x-axis in the figures represents the number of cores allocated to the co-scheduled programs, while the y-axis indicates the relative throughput normalized to that of time-shared scheduling with exclusive solo runs. When cache/bandwidth partitioning features are enabled, we select the best partitioning setup for the given core affinity policy through exhaustive search. The figures demonstrate that the number of cores assigned to the co-scheduled jobs significantly impacts co-run throughput. Simultaneously, the effect on throughput may vary depending on the specific job mix, the chosen affinity policy (compact or distributed), and the cache/bandwidth partitioning configurations.

4.3 Implementation

The insights gained from the previous section motivates us to introduce a comprehensive approach that simultaneously optimizes co-scheduling decisions, NUMA-aware resource assignments, and the cache/bandwidth partitioning setup.

Following on the mathematical formulation described in Section 3.1, we perform throughput-oriented optimization for NUMA systems. Furthermore, as described in Section 3.2, our reinforcement learning-based approach comprises distinct offline phases, namely *profiling* and *training*, along with an online *optimization* phase. The following section delve into the implementation specifics of our reinforcement learning-based solution.

4.3.1 RL Implementation Details

In this section, the components of reinforcement learning, as mentioned in Section 3.3, have been applied to address the resource management challenges specific to NUMA systems.

1. **Agent:** Following on the benefits of Soft Actor-Critic Method as highlighted in Section 2.4.3, the agent has been configured with *soft actor-critic for discrete action settings* [22].
2. **Environment:** In this work, the environment consists of the NUMA system along with its hardware features such as NUMA allocation policies and resource partitioning features for cache/bandwidth.
3. **State:** The state represents the current situation of the system. As described in Section 3.3, the state $s \in \mathbb{R}^{\mathcal{W}\times(n+m+p)}$, where n, m, and p are the number of job features, hardware configurations, and scheduling decisions, respectively. In this context, $n = 12$, signifying the utilization of 12 job features (extracted from hardware performance counters) detailed in Table 4.2. The value $m = 7$, which includes attributes related to resource allocation: (1) selected core allocation policy (*compact/distributed*), (2) selected memory allocation policy (*first touch/round robin/local alloc*), (3) allocated number of cores, (4) allocated number of cache ways to each socket ($\times 2$), and (4) allocated memory bandwidth to each socket ($\times 2$). Moreover, the agent needs to retain two scheduling details for each job, specifically: (1) the co-schedule ID, and (2) whether the job has been executed. These scheduling decision parameters assist the agent in understanding the overall state of the co-execution, resulting in $p = 2$. Additionally, $\mathcal{W}$ represents the job window size, which we can modify according to the evaluation requirements.
4. **Action:** The agent must execute two types of actions: (1) **Schedule-level policies**: As explained in Section 4.1.1, the agent selects the core mapping policy from two alternatives: *compact* or *distributed*. Regarding the memory mapping policy, the options include *first-touch*, *round-robin*, and *local-alloc* (specified by the `numactl` command). (2) **Job-level assignments**: Once the schedule-level policies are determined, the agent proceeds to decide resource allocations in terms of core count, the number of cache ways, and memory bandwidth. For core counts, the

choices are 3, 6, 12, 18, and 21 (out of 24 cores). For Cache allocation we explore five options: 4, 8, 12, 16, and 22 (out of 22 ways on 2 sockets). Memory bandwidth allocation is constrained to five options: 20%, 40%, 60%, 80%, and 100%. During cache and memory bandwidth allocation, the selected cache ways and memory bandwidth are distributed either in a *balanced* manner or *proportionally* (based on the core allocation ratio) to each socket. As mentioned in Section 4.4.1, our target system has two sockets. In addition to these actions, a *skip* option is available to defer the current job to the end of the job window, streamlining the job selection process. Thus, the exploration space encompasses 257 unique actions [(*schedule-level:* 2×3) + (*job-level:* $5 \times 5 \times 5 \times 2$) + *skip*].

5. **Reward:** As detailed in Section 3.3, our reward functions fall into two categories: (1) *intermediate*, designed to provide appropriate rewards by considering job characteristics and resource allocations, and (2) *final*, dependent on the overall throughput improvement of the job mix. The exact definitions of these reward functions are outlined in Table 4.5. Note that the final reward signal is exclusively provided during the execution of the co-schedule on the system.

4.4 Evaluation Setup

4.4.1 Target Platform

Table 4.1: Target Evaluation Platform Details for NUMA System

Name	Remark
CPU	Intel(R) Xeon(R) Silver 4116 x2 sockets
Operating System	Ubuntu 20.04.4 LTS, Kernel Version: 5.13.0-22-generic
Software	Gcc/Gfortran Version: 9.4.0, Numactl Version: 2.0.12-1, Rdtset Version: 3.2.0, Python Version: 3.8.10

The specifications of our evaluation platform are provided in the Table 4.1. Our system comprises two processors equipped with emerging cache and bandwidth partitioning features, namely Intel CAT/MBA, which can be controlled using the `rdtset` command [32]. In our assessment, we apply the Intel CAT cache partitioning feature exclusively to the last level caches. While the accuracy and effectiveness of these partitioning features can vary based on the CPU generation, as indicated in a recent study [77], our RL agent adapts and learns their performance impact during

Table 4.2: Collected Hardware Performance Counters for NUMA system

Statistics
duration_time, task-clock, context-switches, cpu-cycles, instructions, page-faults, branch-misses, L1-dcache-load-misses, L1-icacheload-misses, LLC-load-misses, dTLB-load-misses, iTLB-loadmisses

the training process. The affinities for core/memory mapping are managed using the `numactl` command [38]. Our implementation is executed in Python, leveraging several standard libraries. The reinforcement learning environment is constructed using the gymnasium Python library [27]. For the agent implementation, we employ the PyTorch library [61] to build the deep neural networks for the Soft Actor-Critic Method [22]. Additionally, scikit-learn is used for performing supplementary data pre-processing and feature engineering [62].

We gather hardware performance counters to profile and characterize the applications, employing the Linux perf command [63]. The hardware performance counters collected through the perf command are detailed in Table 4.2. These metrics play a crucial role in evaluating application characteristics, including compute intensity, memory intensity, cache friendliness, and other relevant aspects, which are essential for our approach.

4.4.2 Evaluation Workloads

We employ the `Parsec` benchmark suite [9] and the `NAS Parallel` benchmark suite [7] in our evaluations, as they are widely utilized in studies involving multi-/many-core processors. The specific programs used in our evaluations are listed in Table 4.3.

In our assessment, we initially set the job window size ($\mathcal{W} = 6$) and later vary the size to evaluate the impact of window size selection. Similarly, we fix the maximum concurrency ($\mathcal{C}_{max} = 4$), meaning that at most 4 jobs can be co-located simultaneously. Note that the concurrency may be less than 4, depending on decisions made by the

Table 4.3: Evaluation Benchmarks for NUMA System

Benchmarks Suite	Applications
Parsec	barnes*, cholesky*, fft, fmm, lu_cb, lu_ncb, ocean_cp, ocean_ncp, raytrace*, water_nsquared*, water_spatial
NAS Parallel	BT*, CG, DC*, EP, FT, IS*, MG, SP

Table 4.4: Tested Job Mixes for NUMA System ($\mathcal{W} = 6$)

Name	Jobs
Q1	fmm, FT, EP, fft, CG, lu_ncb
Q2	IS*, lu_ncb, EP, CG, SP, cholesky*
Q3	fft, MG, IS*, lu_ncb, lu_cb, EP
Q4	fmm, IS*, MG, FT, SP, fft
Q5	CG, MG, FT, fmm, DC*, water_nsquared*
Q6	lu_ncb, water_spatial, fft, DC*, fmm, MG
Q7	ocean_ncp, CG, fmm, barnes*, ocean_cp, lu_ncb
Q8	IS*, MG, FT, CG, DC*, ocean_ncp
Q9	MG, EP, IS*, fft, DC*, water_nsquared*
Q10	DC*, barnes*, cholesky*, lu_ncb, CG, fmm
Q11	lu_cb, MG, barnes*, cholesky*, fmm, MG
Q12	raytrace*, cholesky*, DC*, BT*, fft, lu_ncb

agent. For offline training, we exclude 7 programs marked with * in Table 4.3 and utilize the remaining 12 programs. The exclusion is aimed at assessing the generalization capability of our approach to unseen applications.

We generate 16 distinct job queues for agent training, each containing $\mathcal{W}$ programs randomly chosen from the pool of 12 programs. For online inference, we evaluate our approach by randomly creating 12 job queues, sampling jobs from all 19 available programs. The specific job mix selections for $\mathcal{W} = 6$ are detailed in Table 4.4. Programs marked with * are those not encountered during the training phase.

4.4.3 Training/Inference Setup

Table 4.5 outlines the setup for the reward function and the agent in this evaluation. As described in Section 3.3, we employ two types of rewards: (i) *intermediate* reward rw_i and (ii) *final* reward rw_f. The intermediate reward assesses the resource allocation for a chosen job, assessed before launching the job using the associated profile, offering higher rewards for appropriate resource assignments (e.g., allocating more memory bandwidth to a memory-intensive application). Conversely, the final reward corresponds to the measured throughput improvement achieved over time-sharing executions, determined only after completing the co-execution of a job mix. *Note that the reward function can be tailored to optimize for various factors such as energy efficiency, application slowdown, and fairness by modifying the definitions provided in the table.*

In the Table 4.5, *CoreAllocRatio*, *CacheAllocRatio*, and *BWAllocRatio* represent

Table 4.5: Agent and Reward Function Setups for NUMA System

Type	Setups
Reward Function	**Intermediate:** $rw_i = CoreAllocRatio * (ScaleFactorRatio^2 + DurationRatio^2) + (CacheAllocRatio + BWAllocRatio)$ $*$ $L3CacheMissesRatio^2$ **Final:** $rw_f = (SoloRunTime/CoRunTime - 1) \times 100$
Agent	[**# of neurons in the input layer**]: $\mathcal{W} \times (n + m + p)$, [**State Parameters**]: $n = 12, m = 7, p = 2$ [**# of hidden layers for each Network**]3, [**# of neurons in each hidden layer**]: 256/256/256, [**# of neurons in the output layer**]: 257, [**Layer NW**]: Fully connected, [**Activation function**]: Rectified Linear, Softmax

hardware parameters denoting (i) the ratio of allocated cores to the total available cores, (ii) the ratio of allocated cache ways to the total available cache ways, and (iii) the ratio of allocated memory bandwidth to the total available memory bandwidth, respectively. Additionally, *ScaleFactorRatio*, *DurationRatio*, and *L3CacheMissesRatio* serve as job-specific profile parameters with the following definitions: (i) *ScaleFactorRatio* indicates the ratio of the current job's scale factor to the mean scale factor of the job window, where the scale factor is defined as the ratio of running the job on a single core to running it on all available cores; (ii) *DurationRatio* is the ratio of the solo-run execution time of the current job to the mean solo-run execution time of the job window; and (iii) *L3CacheMissesRatio* represents the ratio of L-3 cache misses for the current job to the mean L-3 cache misses of the job window.

As previously mentioned, the agent is configured with the soft actor-critic method for discrete settings [22], and the details are outlined in Table 4.5, where $\mathcal{W}$ represents the window size. This method requires three separate networks for function approximations: one actor network and two critic networks. Specific details about the update rules for each network can be found in the work by Christodoulou [22].

The outlined procedure takes a state vector as input and provides an action to be taken. At the initial time step of an episode, the schedule-level policies are defined. Subsequently, each time step of the training episode determines all required actions for resource allocation for a particular job. This training process is designed to converge towards the global optimum. After completing the training procedure, we conduct tests in the evaluation mode using the trained models.

4.5 Experimental Results

In this section, we present the results of our experiments to evaluate the effectiveness of our approach in optimizing overall throughput across distinct job queues ($\mathcal{Q}1 - \mathcal{Q}12$). Initially, we compare our approach with various alternative scheduling methodologies and resource allocation methods. Following that, we evaluate the ability of the agent to identify appropriate core/memory affinity policies. Subsequently, we investigate the influence of the window size ($\mathcal{W}$) on throughput improvement. Lastly, we present the scheduling overheads associated with our approach.

4.5.1 Throughput Comparison

In our comparative analysis with state-of-the-art methods, we specifically compare our approach with the work conducted by Saba et al. [71], denoted as *OCRP-ML*, due to its relevance to our research focus. It's important to note that while their approach shares similarities with ours, it does not address cache/memory bandwidth partitioning, and the maximum job concurrency is restricted to two. Additionally, in our evaluations, we do not employ the power capping feature associated with their approach.

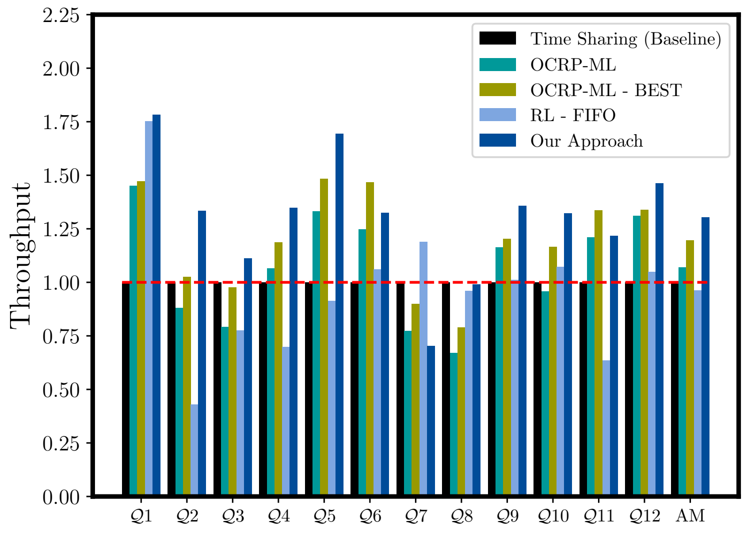


Figure 4.7: Throughput Comparison among Different Scheduling and Resource Allocation Methods for NUMA System ($\mathcal{W} = 6$, $\mathcal{C}_{max} = 4$)

To assess the effectiveness of our approach, we compare it against various scheduling and resource allocation methodologies, including:

1. **Time Sharing (Baseline):** Jobs in the given queue are executed with full system resources without co-scheduling.
2. **OCRP-ML:** We utilize the framework proposed by Saba et al. [71] to evaluate job co-scheduling and resource partitioning with $\mathcal{C} = 2$.
3. **OCRP-ML - BEST:** This method assesses the theoretical maximum throughput for the work of Saba et al. [71] with $\mathcal{C} = 2$. An exhaustive search is employed to identify the optimal combination of job-sets and resource allocations.
4. **RL - FIFO:** Emphasizing the significance of job selection, we evaluate our reinforcement learning (RL) approach with the job selection feature disabled and $\mathcal{C} \leq \mathcal{C}_{max}$.
5. **Our Approach:** We assess our proposed reinforcement learning-based approach, which co-optimizes job selections, core/memory mapping, and resource assignments with $\mathcal{C} \leq \mathcal{C}_{max}$.

Figure 4.7 illustrates the throughput comparison for various scheduling and resource allocation methods. The X-axis represents job queues used for evaluations (AM = Arithmetic Mean), as indicated in Table 4.4, while the Y-axis represents the relative throughput normalized to that of *Time Sharing* for each job queue. In this comparison, we set $\mathcal{W} = 6$ as mentioned before.

Overall, our approach consistently outperforms all other scheduling methods. Compared to the *Time Sharing*, our approach achieves a throughput improvement factor of 1.303 on average. Furthermore, we observe a maximum throughput improvement of up to 1.781 times. In contrast to *OCRP-ML* methods, where cache/memory bandwidth partitioning is not available and concurrency is limited, our approach demonstrates a distinct advantage. Lastly, by emphasizing the critical role of job-set selection quality in co-scheduling, the use of the *RL - FIFO* scheduling policy restricts the agent from achieving higher performance.

4.5.2 Validation of Core/Memory Affinity Choices

In this analysis, we assess whether the designated job sets are assigned with optimal core/memory affinities, referred as NUMA policies, for each job queue. Figure 4.8 presents this validation by comparing the throughput across different methods for each job queue. The X-axis denotes the distinct job queues referenced from Table 4.4, while

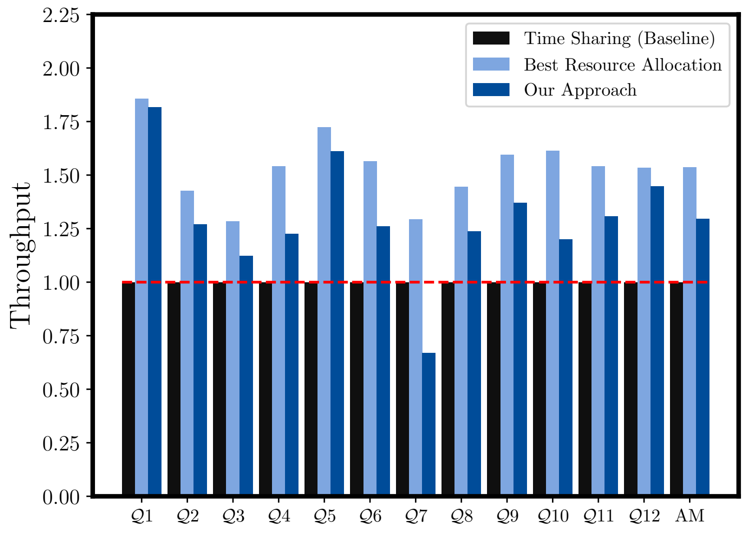


Figure 4.8: Throughput Comparison against the Optimal NUMA Policy Selection ($\mathcal{W} = 6$, $\mathcal{C}_{max} = 4$)

the Y-axis illustrates the relative throughput normalized to the *Time Sharing* method. The *RL - Optimal NUMA Policy* approach involves executing the jobs chosen by the RL agent with corresponding resource assignments utilizing the optimal core/memory affinity, determined through an exhaustive search for cores and memory mapping policies (e.g., compact with first touch).

Overall, our approach achieves nearly optimal throughput for nearly all cases, with an average throughput degradation (or room for improvement) of only 6.38% compared to *RL - Optimal NUMA Policy*. It's worth noting that further improvements in throughput can be realized by adjusting the reward function accordingly. In the current implementation, an intermediate reward is designed on a job-wise basis. However, adopting a global perspective that considers all co-located jobs could enhance the core/memory mapping affinity selections. This is because the potential affinity selections for one job are constrained by those of the other concurrently scheduled jobs.

4.5.3 Scaling Scheduling Attributes

In our approach, we kept $\mathcal{W} = 6$ as the fixed window size and $\mathcal{C}max = 4$ as the maximum concurrency. In this section, we assess the performance of our approach for

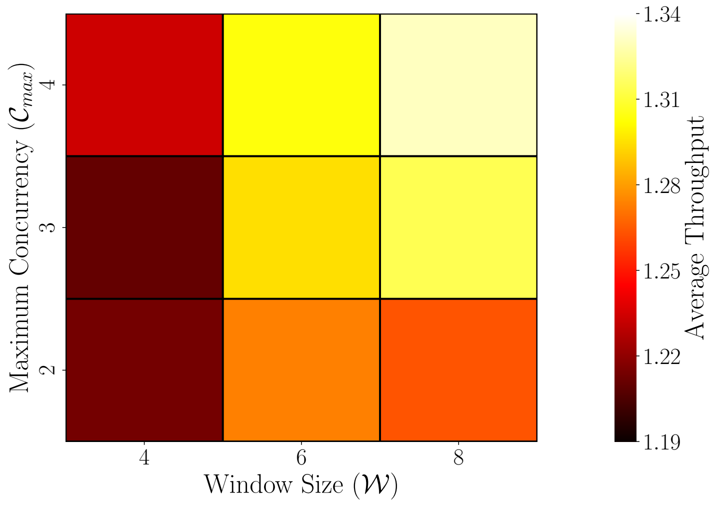


Figure 4.9: Average Throughput comparison for varying values of window size ($\mathcal{W}$) and maximum concurrency ($\mathcal{C}_{max}$) for NUMA System

varying window sizes and concurrency limits. We explore different values for both scheduling attributes, setting $\mathcal{W}$ in the range $[4, 6, 8]$ and $\mathcal{C}max$ in the range $[2, 3, 4]$. Figure 4.9 illustrates the comparison of average throughput for different combinations of $\mathcal{W}$ and $\mathcal{C}_{max}$. The x-axis represents the window size ($\mathcal{W}$), and the y-axis depicts the maximum concurrency ($\mathcal{C}max$). The color-coded visualization indicates the average throughput for each $\mathcal{W} - \mathcal{C}max$ setup. These evaluations are conducted using the same set of jobs listed in Table 4.4. Since there are a total of 72 jobs (6×12) in this table, we form similar job queues using $\mathcal{W} = 4$ and $\mathcal{W} = 8$. The observed trend indicates that our approach scales effectively with the scheduling attributes, displaying a gradual improvement in average throughput with scaled values of $\mathcal{W}$ and $\mathcal{C}_{max}$.

4.5.4 Scheduling Overhead

Finally, we present the time and memory overheads of our scheduling approach. The time overhead of online inference, compared to each job execution time, averages *only 1.06%* across our workloads. The model training time falls within the range of approximately 10-12 hours, required *only once per system*. During inference, the memory overhead of our RL agent is *only 13MiB*, which is negligible when considering the memory capacity of modern systems.

5 Hierarchical Resource Management on Modern GPUs

In this chapter, we focus on introducing "*Hierarchical*" resource management on modern GPUs[1]. As discussed in Section 2.3.2, modern GPUs support various resource partitioning features such as NVIDIA Multi-Process Service and NVIDIA Multi-Instance GPU. We introduce the setup for mixing these features hierarchically and co-locating multiple jobs on the GPU. We quantify the benefits of using such resource partitioning setups using empirical measurements. Nextly, we will delve into further details about the implementation of our reinforcement learning-based solution, specifically targeting *hierarchial* resource management on modern GPUs. Finally, we evaluate our methodology with experimental evaluations.

5.1 Overview

Figure 5.1 illustrates the architecture of a modern GPU, focusing on the hierarchical partitioning features on modern GPUs. As described in Section 2.3.2, the focus is on the features found in the NVIDIA Ampere architecture [54], which includes MIG [55] and MPS [52] functionalities.

GPU partitioning can be achieved through the MIG feature, dividing a GPU into one or more GIs (GPU Instances) at the granularity of GPC. Each GI can then launch one or more CIs (Compute Instances), occupying GPCs within the GI in a mutually exclusive manner. A user can select a CI and run a program on it. Each GI owns the same number of LLC/HBM blocks as GPCs, providing private and isolated resources accessible only by the CIs on the GI. However, as described in Section 2.3.2, MIG's coarse-grained physical partitioning has limitations: (1) one GPC must be disabled when the feature is enabled; (2) it is configurable only when no program is running; and (3) the partitioning choices are limited to 19 variants in the current implementation, lacking support for certain configurations, such as dividing 7 GPCs into 2 GPCs and 5 GPCs.

[1]The results discussed in this chapter have already been published in the Proceedings of IEEE International Conference on Cluster Computing 2023 [72]. All illustrations used in this chapter are *re-drawn* versions of the figures from the published work.

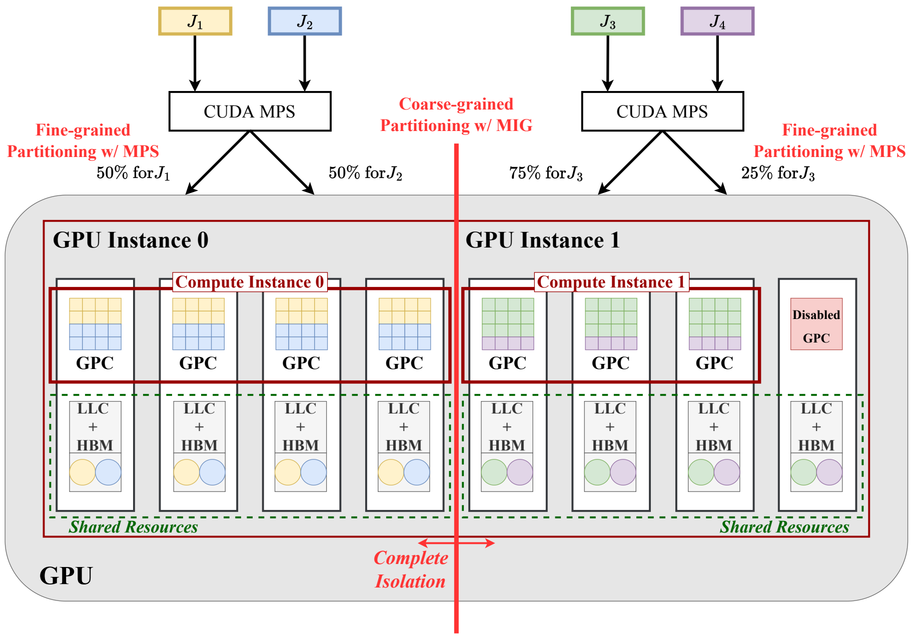
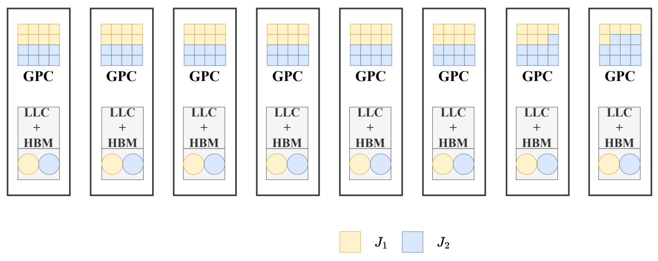


Figure 5.1: Modern GPU architecture with Hierarchical Resource Partitioning

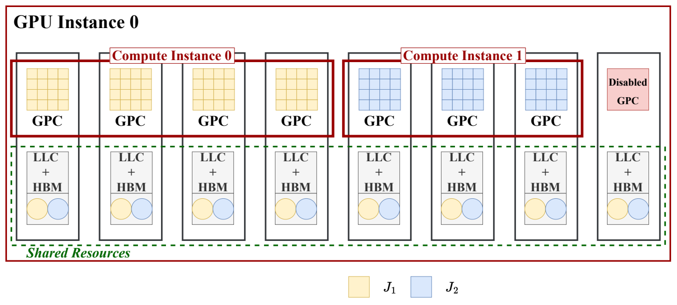
Additionally, the MPS feature allows further partitioning at the granularity of SM, either within each CI or the entire GPU if MIG is not applied. While MPS provides more flexibility and finer-grained partitioning than MIG, it lacks controls for quality of service, such as shared resource partitioning. *Thus, MIG is suitable for setting up shared memory resource partitioning/isolation to mitigate interference impact, while MPS is useful for flexibly assigning compute resources to balance the performance of all co-located programs, surpassing the capabilities of CI-level partitioning.*

The combination of MIG and MPS features presents various partitioning variations, as illustrated in Figure 5.2. Figure 5.2a and Figure 5.2b illustrate the first two options which involve not partitioning memory resources but sharing them among all co-located applications. These options are beneficial when co-located applications require *complementary resources*. For instance, one application may be *compute-bound*, not fully utilizing available memory bandwidth, while another may be *memory-bound*, requiring only a small subset of available compute resources. The MPS-only option offers advantages over MIG-only shared memory, allowing more flexible and fine-grained compute resource allocations, and it avoids the need to turn off 1 out of 8 GPCs, as required by MIG (for A100 GPUs [54]).

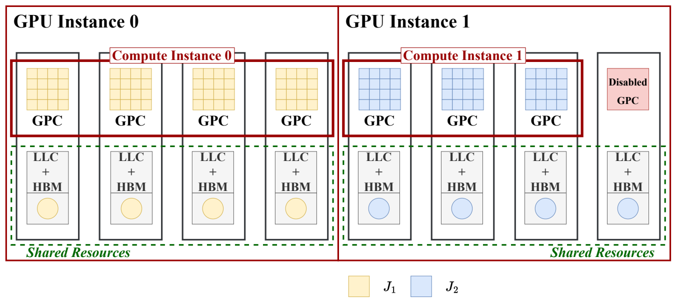
Figure 5.2c illustrates the option designed to mitigate *shared resource conflicts* among co-located applications. This interference-free option is particularly effective for *not*



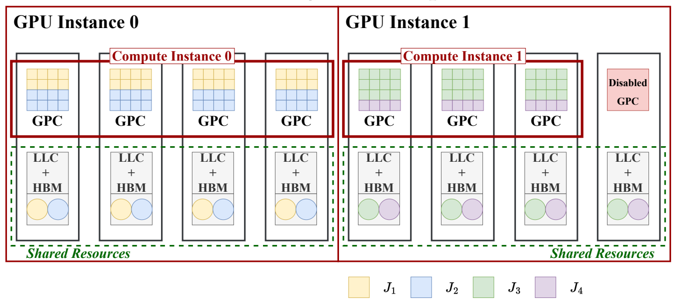
(a) MPS Partitioning w/ Shared Memory



(b) MIG Partitioning w/ Shared Memory



(c) MIG Partitioning w/ Complete Isolation



(d) MIG + MPS Hierarchical Resource Partitioning

Figure 5.2: Resource Partitioning Variants for GPU using MIG and MPS

well-scalable applications, limiting both compute and bandwidth resources on the GPU simultaneously. As Amdahl's law suggests [3], scalability is constrained by a program's parallelism or the overhead of parallelization, which holds for GPU applications limited by issues like synchronization overhead or problem size. This scalability limit within a GPU becomes more critical as compute/bandwidth resources grow richer due to advancements in VLSI technology.

Finally, Figure 5.2d is a combination of MIG and MPS, representing a hierarchial case of all the above options. This approach is promising, especially when executing multiple programs concurrently on the GPU, and it is suitable for a variety of program mixes. The first three options are considered extreme setups of this hierarchical partitioning approach. When co-locating more than two programs within a GI, concurrency increases in the MPS while setting the number of CIs to 1, allowing full flexibility of the MPS feature.

5.2 Observations

5.2.1 Throughput v/s MPS Resource Allocation

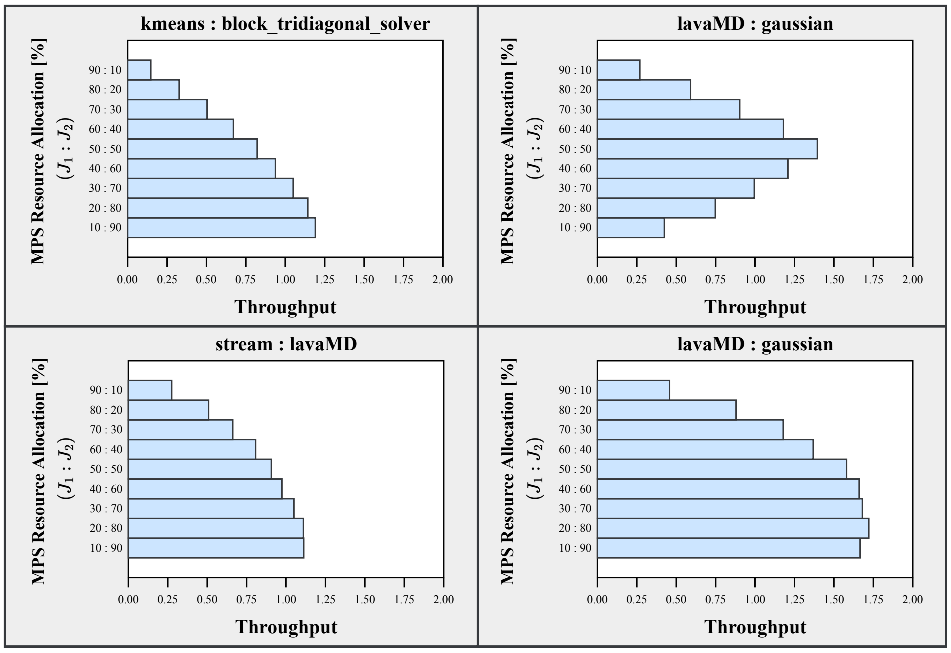


Figure 5.3: Co-scheduling GPU Throughput as a Function of Compute Resource Allocations (MPS Partitioning)

Figure 5.3 illustrates GPU throughput concerning the allocation of compute resources to two co-located jobs across different program mixes. In this assessment, we employ MPS-based partitioning, as depicted in the Figure 5.2a. The Y-axis indicates the ratios of MPS resource allocation to the co-scheduled programs, as indicated in the plot title, while the X-axis denotes the relative throughput normalized to that of time-sharing scheduling. This involves executing the two programs sequentially without resource sharing but with complete allocation of the entire GPU resources.

As demonstrated in Figure 5.3, the optimal allocation of compute resources for co-located programs depends heavily on the specific programs and their characteristics. Notably, for `lavaMD : gaussian`, a balanced allocation achieves the best performance, whereas for the others, a skewed allocation has an advantage over a balanced one, exhibiting a unique optimal allocation point. Given such varying optimal allocations for different program mixes, we conclude that *compute resource partitioning features need to be fine-grained and flexible to allow for precise tuning of the allocation setup, with MPS being more preferable for this purpose.*

5.2.2 Benefit of Bandwidth Partitioning

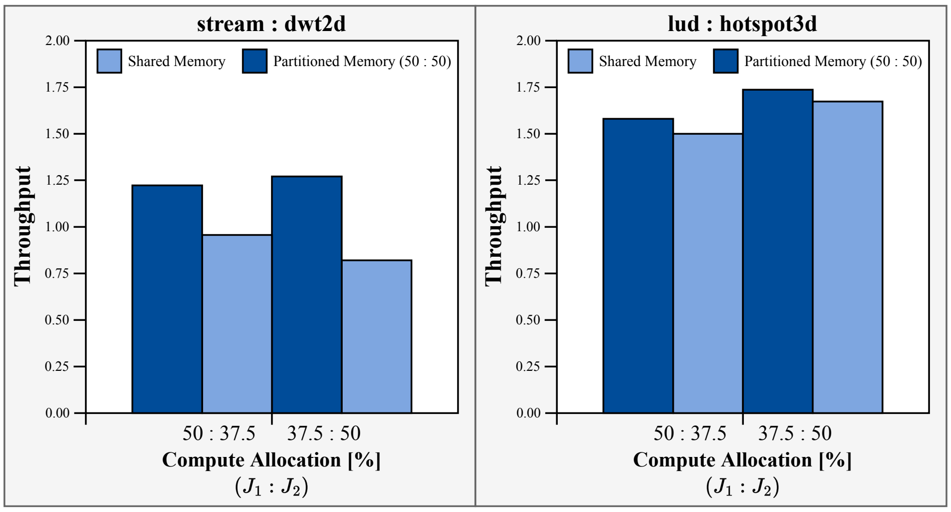


Figure 5.4: Performance Benefit of Bandwidth Partitioning on GPU

Figure 5.4 illustrates the affect of memory bandwidth resource partitioning while utilizing the two distinct MIG options (shared or partitioned) introduced in Figure 5.2b and Figure 5.2c, respectively. The X-axis shows job mixes with varying compute resource allocation rates and memory options (shared or partitioned), while the Y-axis indicates the relative throughput normalized to that of time-sharing scheduling. For evaluating the impact of memory bandwidth partitioning, we configure the exact same resource allocation for both of the memory options: shared and partitioned. As while using MIG, one GPC needs to disabled, resulting in a total compute resource allocation percentage of 87.5% in each case.

We observe a significant throughput improvement when partitioning memory bandwidth, thereby mitigating interference among the co-located programs. Therefore, *depending on the specific job mix, it is preferable to partition/isolate shared memory resources to mitigate interference impact, and only the MIG feature is useful for this purpose.*

5.2.3 Throughput v/s Partitioning Variants

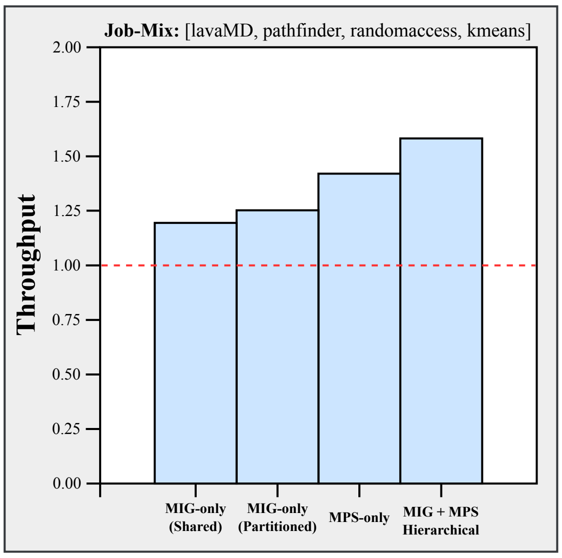


Figure 5.5: Performance Comparison for Different GPU Partitioning Variants

Figure 5.5 illustrates a comparison of various resource partitioning variants described in Figure 5.2. The horizontal axis indicates the various partitioning variants introduced in Figure 5.2, while the vertical axis represents the relative throughput normalized to that of the aforementioned time-sharing scheduling. The job-mix of four programs used for co-scheduling is indicated at the top of the plot, with optimal pairs selected for each partitioning variant.

For the *MPS Only* option, the best compute resource allocation [%] is chosen for the two co-located programs. In the *MIG Only* options, each co-located application is assigned to one of the 4GPC or 3GPC CIs, optimally selected to maximize throughput.

The *MIG+MPS Hierarchical* is a combination of these options. We co-locate all four programs simultaneously on the GPU, initially partitioning it into 4GPCs and 3GPCs with the MIG feature. Then, each co-located program is assigned to one of them with optimal compute resource allocations [%] designated by the MPS feature. Note that we exhaustively search for optimal setups and job pair selections for all the above options. As depicted in the figure, the combination of the two different partitioning features in a hierarchical manner outperforms other partitioning variants. *Thus, hierarchically mixing MIG and MPS helps to maximize the advantages derived from their utilization.*

5.3 Implementation

As discussed in previous sections, hierarchical combination of MIG and MPS is effective in improving the overall throughput of GPUs. For achieving improved performance, it is important to carefully choose the partitioning setup based on the characteristics of the co-located jobs. Additionally, it is vital to select complementary jobs from the job queue for co-location, as job selection can significantly impact the overall system performance. In this work, we co-optimize while targeting both co-scheduling and resource partitioning setup. In this section, we discuss further about our reinforcement learning-based solution and map it to the hierarchial resource management on GPUs.

Following on the mathematical formulation described in Section 3.1, we perform throughput-oriented optimization for GPUs. Additionally, as described in Section 3.2, our reinforcement learning-based approach consists of offline phases: *profiling* and *training*, and an online *optimization* phase. Subsequently, we provide the implementation details for our reinforcement learning-based solution as follows.

5.3.1 RL Implementation Details

Reinforcement learning components have been described in Section 3.3. In this section, these components have been mapped to the resource management problem for GPUs.

1. **Agent:** Following on the benefits of Q-learning as highlighted in Section 2.4.3, we make use of deep-q-learning in this work. The agent has been configured with *duelling double deep Q network*. The choice of this network is based on the benefits outlined in two separate works by Hasselt et al. [86] and Wang et al. [89].

2. **Environment:** In this work, the environment consists of the GPU along with its hardware features such as resource partitioning via MIG and MPS.

3. **State:** The state represents the current situation of the GPU. As described in Section 3.3, the state $s \in \mathbb{R}^{\mathcal{W} \times (n+m+p)}$, where n, m, and p are the number of

job features, hardware configurations, and scheduling decisions, respectively. In this context, $n = 12$, signifying the utilization of 12 job features (extracted from hardware performance counters) detailed in Table 5.2. The value $m = 3$, encompasses resource allocation for each job, including: (1) allocated SM count, (2) allocated Memory in GiB, and (3) memory option (shared or isolated). Further, the agent is required to store two scheduling details for each job, namely: (1) co-schedule ID, and (2) whether the job has been executed. These scheduling decision parameters helps agent to understand the overall state of the co-execution, hence $p = 2$. Additionally, $\mathcal{W}$ is the job window size, which we can adjust based on the evaluation requirements.

4. **Action:** As the agent encounters a job, it has the option to either select one of 27 resource partitioning schemes and assigning it to the job or *skip* the job. The skip mechanism has been incorporated to facilitate the job selection process. Once the co-schedule has been generated, the agent needs to take action to *execute* the co-schedule for evaluating its performance. Hence, we have in-total 29 actions in this implementation.

5. **Reward:** As mentioned in Section 3.3, we have two categories of reward functions: (1) *intermediate*: designed to offer suitable rewards considering job characteristics and resource allocations (such as SM count and Memory), and (2) *final*: based on the overall throughput improvement of the job-mix. The exact definitions of the reward function has been described in Table 5.5. Note that the final reward signal is only provided when the co-schedule is being executed on the GPU.

5.4 Evaluation Setup

In this section, we describe our evaluation setups including our target platform, workload selections, neural network configurations, compared methods, and partitioning variants.

5.4.1 Target Platform

Table 5.1 lists the target platform used for evaluating our approach. As mentioned before, we utilized an A100 GPU and applied the MIG and MPS features to it. Our reinforcement learning-based solution has been implemented in Python using multiple standard python libraries. We build our reinforcement learning environment using the gymnasium python library [27]. For implementing the agent, we use the PyTorch library for implementing the deep neural networks for Q-learning [61]. Further, we use

Table 5.1: Target Evaluation Platform Details for GPU

Name	Remark
GPU	NVIDIA A100 40GB PCIe 250W TDP
Operating System	Ubuntu 20.04.4 LTS, Kernel Version: 5.4.0-137-generic
Software	CUDA Version: 11.6, Driver Version: 510.108.03, Python Version: 2.7.18

Table 5.2: Collected Hardware Performance Counters for GPU

Statistics
Duration, Memory [%], Elapsed Cycles, Grid Size, Registers Per Thread, DRAM Throughput, L1/TEX Cache Throughput, L2 Cache Throughput, SM Active Cycles, Compute (SM) [%], Waves Per SM, Achieved Active Warps Per SM

scikit-learn for performing additional data pre-processing and feature engineering [62]. We collect hardware performance counters to profile and characterize the applications. To this end, we utilize the NVIDIA Nsight compute framework [56]. Table 5.2 lists the collected hardware performance counters by using the framework. These statistics are useful to characterize the applications in terms of compute intensity, memory intensity, parallelism/scalability, memory access pattern, and so forth.

5.4.2 Evaluation Workloads

We use the `Rodinia` benchmark suite [18], a `stream` [23] benchmark, a `randomaccess` [39] benchmark, and the `Quicksilver` mini application selected from the CORAL benchmark suite [44]. These benchmark programs are categorized into *CI (Compute Intensive)*, *MI (Memory Intensive)*, and *US (UnScalable)*, as outlined in Table 5.3. The classification follows a methodology established in a previous study [4]: (1) If the performance degradation caused by a 1GPC run with the private memory option, compared with the full 8GPC run, is less than 10%, we designate it as an UnScalable (US) application; (2) otherwise, if the ratio of `Compute (SM) [%]` to `Memory [%]` exceeds 0.80, we classify it as a Compute-Intensive (CI) application; (3) otherwise, it is categorized as an Memory-Intensive (MI) application.

In our evaluations, we initially set the job window size (W) to twelve and later vary this size to evaluate the impact of window size selection. For offline training,

Table 5.3: Evaluation Benchmarks for GPU along with classification

Class	Benchmarks
CI	lavaMD, huffman*, hotspot3D, hotspot*, heartwall*, bt_solver_A, bt_solver_B, bt_solver_C
MI	lud_A, lud_B, lud_C*, sp_solver_A, sp_solver_B, sp_solver_C, randomaccess, cfd*, gaussian*, stream
US	kmeans, dwt2d, needle*, pathfinder, backprop*, qs_Coral_P1, qs_Coral_P2, qs_NoFission*, qs_NoCollisions

we exclude nine programs marked with an asterisk (*) in Table 5.3 and utilize the remaining 18 programs. The exclusion aims to test the generalization capability of our approach to unseen workloads.

We create 20 different job queues for agent training, each comprising $\mathcal{W}$ programs randomly selected from the 18 programs while ensuring representation from all three categories in the queue. For online inference, we test our approach with various job mixes: (1) *CI-dominant*; (2) *MI-dominant*; (3) *US-dominant*; and (4) *Balanced*. In the *X-dominant* job mix, 50% of applications are from class X (X=CI, MI, or US), and the remaining 50% are from the other classes in a round-robin manner. For example, when $\mathcal{W} = 12$, the *CI-dominant* class comprises 6 CI, 3 MI, and 3 US applications. The *Balanced* job mix selects application classes in a round-robin manner, consisting of 4 CI, 4 MI, and 4 US applications when $mathcalW = 12$. For each job mix category, we create several variants (A, B, and C), assigning applications randomly selected from Table 5.3 to each application class. The specific job mix selections for $\mathcal{W} = 12$ are detailed in Table 5.4. Note that programs marked with * are unseen during training.

5.4.3 Training/Inference Setup

Table 5.5 outlines the configurations employed for the reward function and the agent in our evaluation, incorporating two types of rewards: (i) *intermediate* reward r_i and (ii) *final* reward r_f. The intermediate reward assesses the resource allocation for a selected job, determined before job launch using the associated profile. It yields a higher reward for better resource assignment, such as allocating more memory bandwidth to a memory-intensive application. Conversely, the final reward gauges the measured throughput improvement over time-sharing executions, accessible only after completing the co-execution of a job mix.

In the table, $SmAllocRatio$ and $MemoryAllocRatio$ are hardware parameters, signifying (i) the ratio of allocated Streaming-Multiprocessors to the total count and (ii) the ratio

Table 5.4: Tested Job Mixes per Category for GPU ($\mathcal{W} = 12$)

Category	Name	Jobs
CI-dominant (CIx6, MIx3, USx3)	$\mathcal{Q}1$	huffman*, bt_solver_C, bt_solver_B, hotspot3D, heartwall*, lavaMD, lud_B, cfd*, sp_solver_B, pathfinder, needle*, qs_NoFission*
	$\mathcal{Q}2$	bt_solver_C, heartwall*, lavaMD, huffman*, hotspot*, hotspot3D, cfd*, sp_solver_C, gaussian*, pathfinder, needle*, qs_Coral_P1
	$\mathcal{Q}3$	huffman*, bt_solver_C, hotspot3D, hotspot*, heartwall*, lavaMD, lud_B, stream, sp_solver_C, qs_NoFission*, pathfinder, needle*
MI-dominant (CIx3, MIx6, USx3)	$\mathcal{Q}4$	bt_solver_B, heartwall*, bt_solver_C, lud_B, gaussian*, sp_solver_B, cfd*, sp_solver_C, stream, qs_NoCollisions, pathfinder, qs_Coral_P2
	$\mathcal{Q}5$	heartwall*, hotspot*, bt_solver_B, lud_B, gaussian*, randomaccess, stream, lud_C*, sp_solver_B, qs_Coral_P2, dwt2d, qs_Coral_P1
	$\mathcal{Q}6$	bt_solver_C, huffman*, lavaMD, sp_solver_B, gaussian*, randomaccess, lud_C*, stream, cfd*, qs_NoFission*, needle*, qs_Coral_P1
US-dominant (CIx3, MIx3, USx6)	$\mathcal{Q}7$	heartwall*, hotspot*, hotspot3D, gaussian*, stream, lud_B, pathfinder, qs_NoFission*, qs_Coral_P2, backprop*, qs_NoCollisions, dwt2d
	$\mathcal{Q}8$	bt_solver_C, hotspot3D, lavaMD, stream, cfd*, lud_B, qs_Coral_P1, needle*, kmeans, qs_Coral_P2, qs_NoFission*, qs_NoCollisions
	$\mathcal{Q}9$	lavaMD, hotspot3D, hotspot*, sp_solver_B, lud_C*, randomaccess, qs_Coral_P1, dwt2d, kmeans, needle*, qs_NoCollisions, qs_Coral_P2
Balanced (CIx4, MIx4, USx4)	$\mathcal{Q}10$	lavaMD, huffman*, hotspot3D, bt_solver_C, lud_C*, lud_B, stream, sp_solver_C, qs_NoCollisions, needle*, pathfinder, qs_Coral_P1
	$\mathcal{Q}11$	huffman*, hotspot3D, hotspot*, bt_solver_B, cfd*, lud_C*, stream, gaussian*, qs_Coral_P2, needle*, pathfinder, dwt2d
	$\mathcal{Q}12$	lavaMD, hotspot*, huffman*, heartwall*, sp_solver_C, lud_C*, randomaccess, gaussian*, needle*, pathfinder, qs_NoCollisions, backprop*

Table 5.5: Agent and Reward Function Setups for GPU

Type	Setup
Reward Function	**Intermediate:** $rw_i = (SmAllocRatio \times ComputeRatio + MemoryAllocRatio \times MemoryRatio) \times DurationRatio^2$ **Final:** $rw_f = (SoloRunTime/CoRunTime - 1) \times 100$
Agent	[**# of neurons in the input layer**]: $\mathcal{W} \times (n + m + p)$, [**State Parameters**]: $n = 12, m = 3, p = 2$ [**# of neurons in the output layer**]: $\mathcal{V}$ = 1, $\mathcal{A}$ =29, [**# of hidden layers**]: 3, [**# of neurons in each hidden layer**]: 512/256/128, [**Layer NW**]: Fully connected, [**Activation function**]: Rectified Linear

of allocated memory bandwidth to the total available memory bandwidth, respectively. The parameters *ComputeRatio*, *MemoryRatio*, and *DurationRatio* are job-specific profile parameters characterized as follows: (i) *ComputeRatio*: the ratio of `Compute (SM) [%]` for the current job to the mean `Compute (SM) [%]` of the job window, (ii) *MemoryRatio*: the ratio of `Memory [%]` for the current job to the mean `Memory [%]` of the job window, and (iii) *DurationRatio*: the ratio of solo-run execution time for the current job to the mean solo-run execution time of the job window.

While our current reward function focuses on optimizing for co-run throughput, its adaptability allows for further refinement, incorporating additional parameters like job-specific priorities, scheduling fairness, and energy consumption.

Regarding the agent, it is configured with a double dueling deep Q-network [89], with details provided in Table 5.5. In this network, the Q-value is decomposed into two components: (i) $\mathcal{V}$ value, representing the state's value, and (ii) $\mathcal{A}$ advantage, signifying the benefit of selecting a specific action in the given state. The update rule for the Q-value, utilizing $\mathcal{A}$ and $\mathcal{V}$, is detailed in Wang et al.'s work [89]. Following on prior work [86], we employ two networks based on the same architecture: one for predicted Q-value and the other for target Q-value. As mentioned in Section 2.4.3, training involves using the ϵ-greedy approach, where the parameter ϵ starts at 1 and gradually diminishes until reaching a set point (e.g., 0.01 in our evaluation). This parameter controls the frequency of random actions taken by the agent. Specifically, with a probability of ϵ, the agent randomly selects an action from the entire search space. This iterative procedure aims to converge toward the global optimum. Once the training phase concludes, ϵ is set to 0 to eliminate random actions when utilizing the trained agent in the online phase.

5.4.4 Compared Methods

To evaluate the effectiveness of our approach, we compare various scheduling policies with respect to throughput, application slowdown, and fairness when handling given job mixes. The following methods are examined:

- **Time Sharing (Baseline)**: Jobs in the provided job mix (or queue) are executed using the entire GPU resources exclusively, without any co-scheduling or partitioning.
- **MIG Only ($\mathcal{C} = 2$)**: In line with previous studies [4, 71], we explore the MIG-only option with a concurrency level $\mathcal{C}$ set to 2. The job set selections and assignments are optimized through exhaustive consideration of all possible setups.
- **MPS Only ($\mathcal{C} \leq \mathcal{C}_{max}$)**: The MPS-only option is tested with various concurrency selections ($\mathcal{C} \leq \mathcal{C}_{max}$). The job set selections and resource assignments are determined through an exhaustive search.
- **MIG+MPS Default ($\mathcal{C} \leq \mathcal{C}_{max}$)**: MIG partitioning is selected to maximize the average throughput across $\mathcal{Q}1 - \mathcal{Q}12$. The MPS allocation is set to the *default* mode (which is 100% allocation of the active thread percentage). The job set selections (L_{JS}) are optimized through exhaustive search within the designated concurrency limit and configuration space.
- **MIG+MPS w/ RL ($\mathcal{C} \leq \mathcal{C}_{max}$)**: Our proposed reinforcement learning-based co-optimization of co-scheduling and hierarchical resource partitioning.

5.4.5 Evaluated Resource Partition Setups

Table 5.6 enumerates the partitioning variants examined in the evaluation across different concurrency setups ($\mathcal{C}$). As previously discussed, the variants are specified for *MPS Only* and *MIG+MPS w/ RL*. Additionally, for *MIG Only*, we investigate the two options illustrated in Figure 5.2b and Figure 5.2c to provide a comparative analysis with existing works [4, 71]. In the case of *MIG+MPS w/ Default*, it involves assigning the default active thread percentage over the optimized MIG partitions.

The format for representing partitioning states is defined as follows. Firstly, a GI or the entire GPU is enclosed in square brackets, denoted as [compute resource setup, assigned memory resource]. For the memory resource part, when $\alpha \times 100\%$ of the entire GPU memory bandwidth is assigned, it is expressed as "αm". Regarding the compute resource setup, a CI or an MPS process is enclosed in curly brackets or parentheses, respectively. The number in brackets (denoted as β) signifies the allocated compute resources (i.e., $\beta \times 100\%$ of the GPU total). For example, [β, αm] signifies

Table 5.6: GPU Partitioning Setups for Different Concurrency (See Section 5.4.5 for the Format Definition)

$\mathcal{C}$	For MPS Only	For MPS+MIG w/ RL
2	[(0.1)+(0.9),1m]; [(0.2)+(0.8),1m]; . . . ; [(0.5)+(0.5),1m];	[(0.1)+(0.9),1m]; [(0.2)+(0.8),1m]; . . . ; [(0.5)+(0.5),1m]; [{0.375}+{0.5},1m]; [{0.375},0.5m]+[{0.5},0.5m]
3	[(0.1)+(0.1)+(0.8),1m]; . . . ; [(0.34)+(0.33)+(0.33),1m];	[(0.1)+(0.1)+(0.8),1m]; . . . ; [(0.34)+(0.33)+(0.33),1m]; [{0.375},0.5m]+[(0.1)+(0.9),{0.5},0.5m]; . . . ; [{0.375},0.5m]+[(0.5)+(0.5),{0.5},0.5m]; [{0.375}+(0.1),(0.9){0.5},1m]; . . . ; [{0.375}+(0.5),(0.5){0.5},1m];
4	[(0.1)+(0.1)+(0.1)+(0.7),1m]; . . . ; [(0.25)+(0.25)+(0.25)+(0.25),1m];	[(0.1)+(0.1)+(0.1)+(0.7),1m]; . . . ; [(0.25)+(0.25)+(0.25)+(0.25),1m]; [(0.1)+(0.9),{0.375},0.5m]+ [(0.1)+(0.9),{0.5},0.5m]; . . . ; [(0.5)+(0.5),{0.375},0.5m]+ [(0.5)+(0.5),{0.5},0.5m]; [(0.1)+(0.9){0.375}+(0.1)+(0.9){0.5},1m]; . . . ; [(0.5)+(0.5){0.375}+(0.5)+(0.5){0.5},1m];

the presence of one CI inside the GI, capable of utilizing $\beta \times 100\%$ (or $\alpha \times 100\%$) of compute (or bandwidth) resources. Furthermore, partitions at the same level of the hierarchy are combined with "+" in the format. For instance, [0.375+0.5,1m] represents the 3GPC+4GPC MIG-only partitioning with the shared memory option, whereas [0.375,0.5m]+[0.5,0.5m] signifies the isolated memory option with the same GPC allocations.

5.5 Experimental Results

5.5.1 Throughput Comparison

Figure 5.6 presents a throughput comparison across different methods and diverse workloads. The horizontal axis depicts the executed workloads (AM: Arithmetic Mean), while the vertical axis showcases relative throughput normalized to that of *Time Sharing* for each workload. Throughout the assessment, the maximum concurrency ($\mathcal{C}_{max}$) is consistently set at 4. Generally, the proposed reinforcement learning-based approach

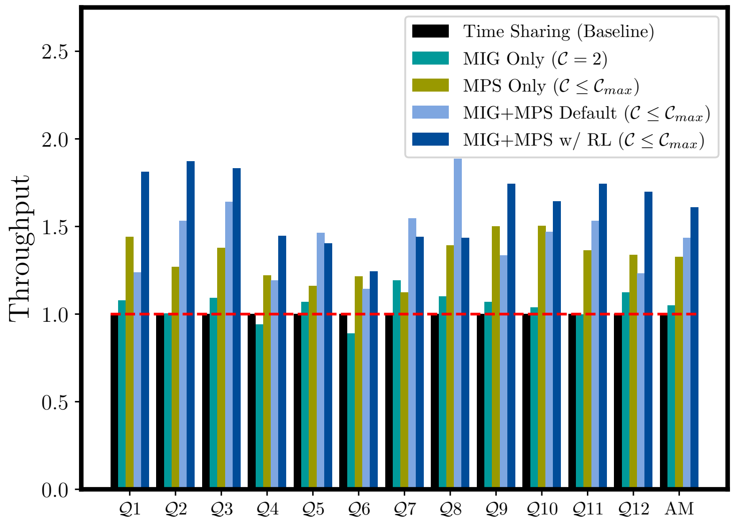


Figure 5.6: Throughput Comparison for GPU Evaluations ($\mathcal{C}_{max}$ = 4, $\mathcal{W}$ = 12)

outperforms all other methods across a wide range of workloads. In comparison to *Time Sharing*, it achieves an average throughput improvement of 1.516 and a peak improvement of 1.873. The *MIG+MPS Default* method is also hierarchical and uses fixed MIG partitioning and default MPS configuration. However, our approach surpasses this option, indicating that hierarchical partitioning should be dynamically adjusted based on the characteristics of co-scheduled jobs. The effectiveness of the *MPS Only* option is limited as it cannot effectively mitigate interference on shared resources among co-scheduled programs. However, when combined with the MIG feature, its effectiveness improves.

5.5.2 Scaling Scheduling Attributes

In the subsequent figure, Figure 5.7 depicts the average throughput relative to the maximum job concurrency ($\mathcal{C}_{max}$) and the window size ($\mathcal{W}$). The vertical axes in these sub-figures showcase the average throughput computed across all 12 job queues ($\mathcal{Q}1 - \mathcal{Q}12$), and the horizontal axes represent the respective attributes: $\mathcal{C}_{max}$ and $\mathcal{W}$. Notably, when scaling $\mathcal{C}_{max}$, $\mathcal{W}$ is maintained at a value of 12, whereas when scaling $\mathcal{W}$, C_{max} is consistently fixed at 4. The figure reveals an upward trend in throughput as these parameters are scaled. This can be attributed to two key factors: (1) our approach

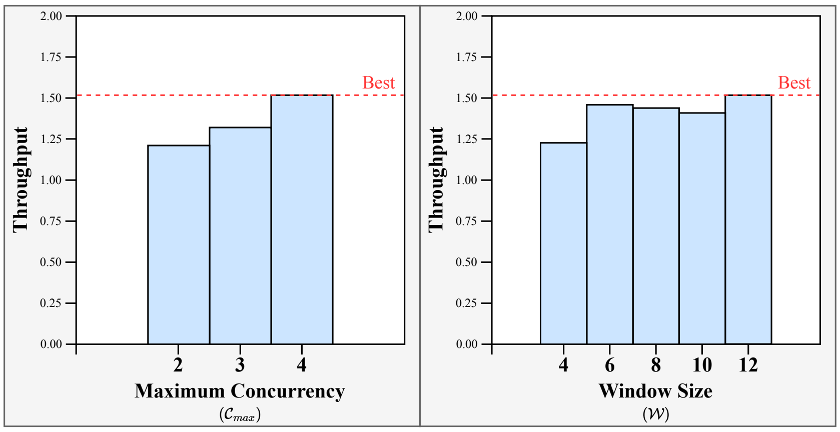


Figure 5.7: Scaling GPU Scheduling Attributes

excels in identifying more optimal co-scheduling groups for larger $\mathcal{W}$ values; and (2) flexible partitioning and shared resource isolation, facilitated by MPS and MIG, allows our co-scheduling to utilize resources more efficiently for higher $\mathcal{C}_{max}$. Ultimately, we determined that $\mathcal{W} = 12$ and $\mathcal{C}_{max} = 4$ offer optimal throughput for our workloads, as further scaling did not yield additional improvements.

5.5.3 Application Slowdown Comparison

Figure 5.8 provides insights into the average application slowdown induced by co-scheduling across various methods and job queues. The X-axis represents the evaluated workloads, while the Y-axis signifies the average application slowdown. The application slowdown ($AppSlowdown$) for a specific job (J) drawn from a given queue ($\mathcal{Q}_i$) is defined as the ratio of its co-scheduled execution time ($CoRunAppTime(J)$) to its solo-run execution time ($SoloRunAppTime(J)$):

$$AppSlowdown(J) = \frac{CoRunAppTime(J)}{SoloRunAppTime(J)}$$

The average application slowdown is then computed across all jobs within the respective queue for each method. Our approach demonstrates an average application slowdown of 1.829 and a best-case slowdown of 1.345. Despite the presence of application slowdowns in co-scheduling scenarios, the approach achieves higher throughput overall, as observed in Figure 5.6. It leverages increased concurrency up to $\mathcal{C}_{max}$, contributing to enhanced total system throughput. Notably, while the average application slowdown for *MIG Only ($\mathcal{C} = 2$)* is smaller than that of other methods, its limited concurrency results in a lower overall throughput. As our approach can

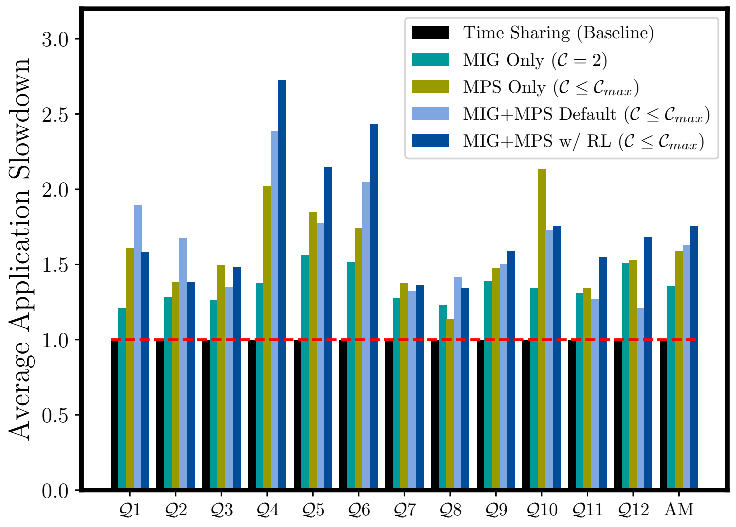


Figure 5.8: Average Application Slowdown Comparison for GPU Evaluations ($\mathcal{C}_{max} = 4$, $\mathcal{W} = 12$)

trade-off the application slowdowns and concurrency in a better way, it achieves higher total system throughput as a consequence.

5.5.4 Fairness Comparison

Figure 5.9 offers a comparative analysis of scheduling fairness across different methods and various workloads. Employing a fairness metric from a previous study [49], we define the fairness metric (*Fairness*) for a given queue ($\mathcal{Q}_i$) as follows:

$$Fairness(\mathcal{Q}_i) = \frac{\min_{J \in \mathcal{Q}_i}(AppSlowdown(J))}{\max_{J \in \mathcal{Q}_i}(AppSlowdown(J))}$$

A higher value is preferable for this metric, with the maximum attainable value being 1. Specifically, when the fairness metric equals one, it implies that the maximum slowdown aligns precisely with the minimum slowdown, signifying that all applications experience an equivalent degree of slowdown. As depicted in Figure 5.9, our approach exhibits fairness comparable to other methods, except for Time Sharing, even though it outperforms them in terms of throughput. Note that the fairness in our approach can be further enhanced by incorporating it into the reward function.

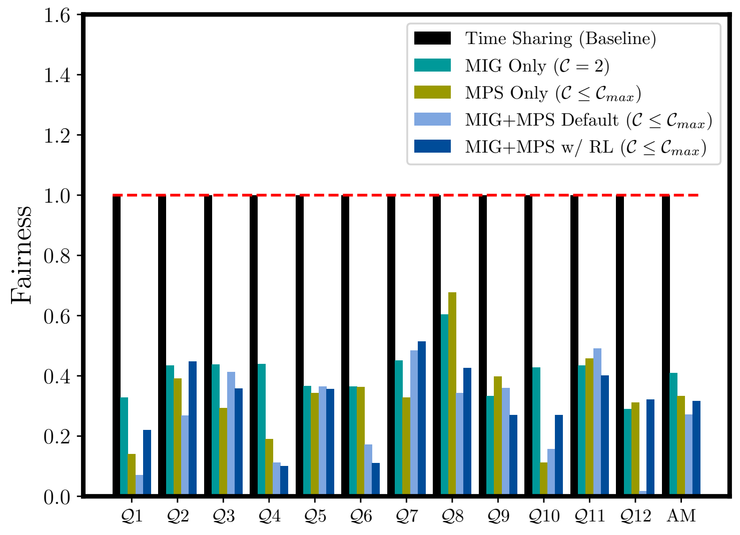


Figure 5.9: Fairness Comparison for GPU Evaluations ($\mathcal{C}_{max}$ = 4, $\mathcal{W}$ = 12)

5.5.5 Scheduling Overhead

Lastly, we report the overhead of our approach in both the online and offline phases. The throughput degradation caused by our online optimization is less than 0.5% on average across our workloads ($\mathcal{W} = 12$), which is negligible compared with the throughput gain, and thus we observe the considerable throughput improvement, as shown in Figure 5.6. As for the offline training time, a key bottleneck arises due to real-time interactions with the system, i.e., continuous benchmark runs. With available MIG/MPS setups for the selected concurrency (let $N_{\mathcal{C}}$ be the number of available setups for $\mathcal{C}$, see also Table 5.6), the maximum count of distinct job selections plus resource assignments is $\sum_{\mathcal{C}=2}^{\mathcal{C}_{max}} \binom{\mathcal{W}}{\mathcal{C}} \times \mathcal{C}! \times N_{\mathcal{C}}$. Here, to assess the maximum, we suppose selecting $\mathcal{C}$ jobs from $\mathcal{W}$ unique jobs and assigning them to $\mathcal{C}$ distinct regions partitioned with a certain MIG/MPS setup chosen from $N_{\mathcal{C}}$ variants. Consequently, for $\mathcal{W} = 12$ and $\mathcal{C}_{max} = 4$, the training overhead could escalate to the order of $10^5 \times t_{avg}$, where t_{avg} signifies the average duration taken for executing a scheduling policy on the system. However, as the agent progressively converges towards optimal policies, it need not explore every conceivable policy within this set. Hence, in our environment, the offline training procedure takes only couple of hours. The overhead is reasonable as the training is required only once for a system.

6 Discussion

In this chapter, we would like provide additional discussions on three aspects: (1) Firstly, we will also explore possible future extension opportunities for our work, (2) Secondly, we would like to propose a potential future work on coordinated resource management for heterogeneous HPC systems, and (3) Lastly, we will present a comparison with other existing methods,

6.1 Extension Opportunities

6.1.1 Hybrid Memories

Our methodology, as described in chapter 4, naturally extends to hybrid memory systems incorporating multiple memory devices. Typically, faster/smaller memories in such systems serve as hardware caches or scratchpads [35], and our approach is versatile for both scenarios. For hardware caches, our agent can learn optimal partitioning/scheduling by incorporating cache statistics from additional memory devices into job profiles. In the case of scratchpads mapped to different NUMA domains [35], extension of our agent can handle memory assignment optimization by including more memory affinity options in decision-making.

6.1.2 Cluster of GPUs

Our approach, as described in chapter 5, is readily adaptable to GPU clusters, as optimizations at the node level naturally extend to clusters and directly influence GPU cluster operations. To achieve this extension, the hierarchical optimization framework outlined in this study must be enhanced by incorporating an additional level of resource assignments at the top, specifically, node/GPU allocations. To facilitate this extension, the vector of job characteristics denoted as J_i must include the numbers of GPUs/nodes requested by the job, information that can be obtained from the corresponding job script. Based on this information, the agent will determine the resource allocations denoted as R_i, which also need expansion to encompass the physical IDs of assigned nodes/GPUs, along with their partitioning states. Moreover, the agent and the reward function must collaboratively address load imbalances introduced by co-scheduling multi-node/-GPU jobs. For instance, a multi-node/-GPU job may be co-located with different jobs on distinct nodes/GPUs, leading to a considerable load imbalance for the job.

We envision two options for this extension: (1) introducing a more extensive and scalable neural network; (2) utilizing a multi-level agent to separately handle system-wide and node-level optimizations, but in a coordinated manner.

For option (1), we will employ a larger and more scalable neural network to enhance training and inference efficiency. Additionally, we will modify the reward function to incorporate multi-node/-GPU optimization considerations. In terms of the agent, this extension transforms the problem into a three-dimensional packing problem. Here, one dimension corresponds to node/GPU assignments, another to the time domain, and the third to the resource assignments within each GPU. Note that the first dimension can also be treated as a hierarchical resource management structure if each node comprises multiple GPUs, mirroring the hierarchical approach used in this work. The additional challenges for the agent include: (1) considering the number of GPUs/nodes each job occupies and the total number available on the cluster when selecting jobs to schedule; and (2) ensuring synchronization of resource partitioning within each GPU across all GPUs to avoid significant load imbalances across all scheduled workloads.

The focus of this work is on overcrowded systems with prolonged queuing times, where there is a constant influx of runnable jobs, a scenario commonly encountered in HPC centers with GPU demand exceeding GPU availability. In such situations, co-locating multiple GPU jobs on the same GPU(s) to maximize throughput, similar to our approach, can be highly efficient. As the system becomes less crowded, conventional scheduling policies like FCFS (First Come First Serve) with backfilling, without co-scheduling, may become more efficient. Therefore, in practice, a policy selection mechanism that adapts to the system state, considering currently running and queuing jobs, becomes essential. Developing such a mechanism is an intriguing research direction and could be a focus of our future studies, along with integrating our approach into existing HPC cluster management tools like Slurm [76].

6.1.3 Other Possible Improvements

In this study, we simplify application input dependencies by using constant inputs for each application throughout the evaluation. In practical HPC scenarios, applications often run with different inputs, altering their behaviors. To address this, future work will explore existing schemes [34, 17] to estimate/compensate for the impact of input changes on application characteristics.

Security concerns in shared node resources during co-scheduling include potential violations of inter-process data protection and quality of service (QoS). Existing solutions for general-purpose computing can handle inter-process data protection vulnerabilities at a low level, such as the OS or firmware. QoS can be managed by developing a robust/fair job selection and resource partitioning mechanism as an extension of our work. Additionally, an interference-aware pricing scheme may be effective in addressing noticeable QoS violations [16].

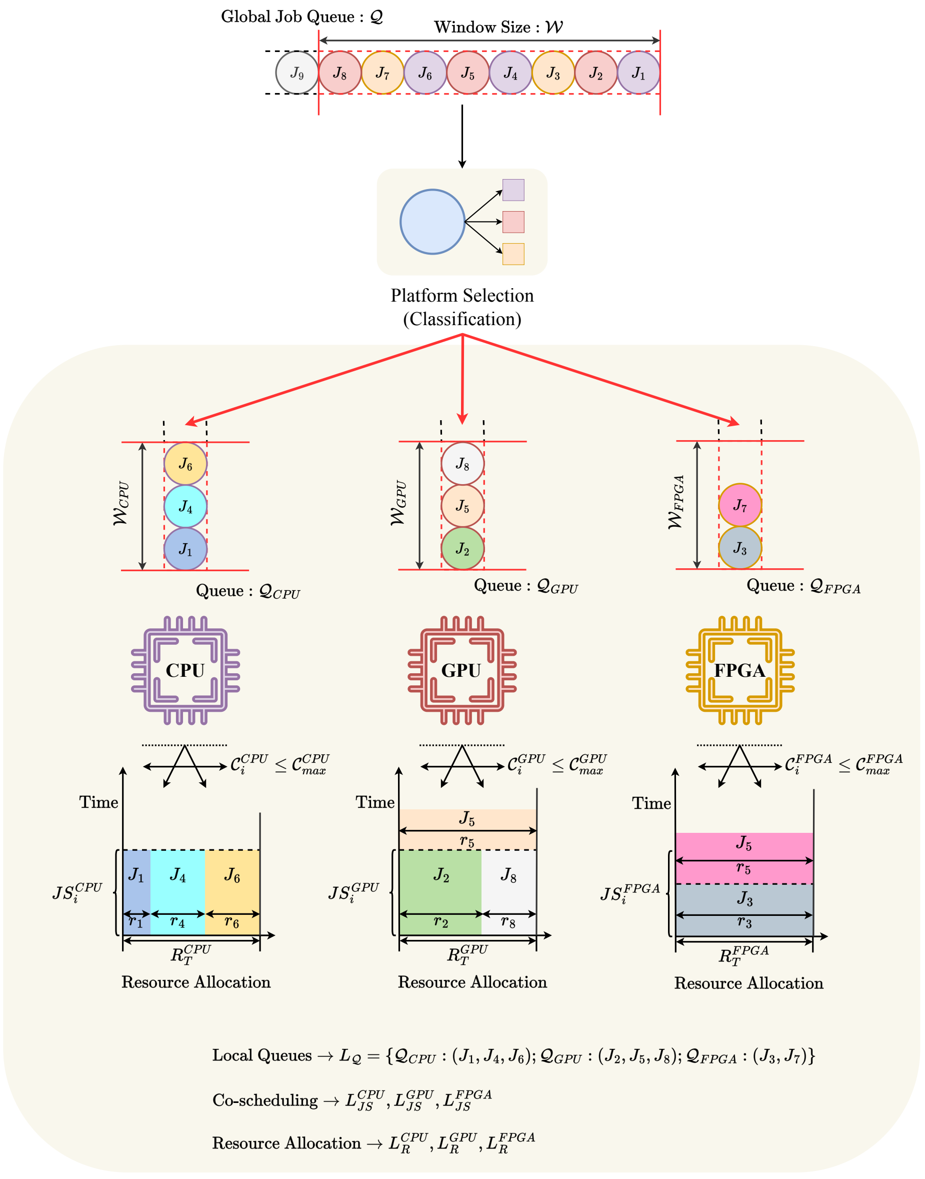
While our focus in this work is on profile-driven static management, where job selections and resource assignments are static decisions in HPC, it is promising to explore dynamic adjustments for some controllable knobs/resources (e.g., Intel CAT and MBA) at runtime. This dynamic fine-tuning can be particularly useful when applications change their behaviors dynamically, and it represents a promising extension/enhancement for our work, potentially through updating our agent or integrating our approach with existing tools like DRLPart [20].

6.2 Coordinated Approach to Heterogeneous HPC Resource Management

As discussed in Chapter 1, modern HPC systems are characterized by increasing heterogeneity, incorporating CPUs, GPUs, and FPGAs within a single node. While our research has predominantly focused on resource management for CPUs and GPUs separately, the proposed methodology has shown efficacy in enhancing platform-specific performance for these components. However, it does not directly address the broader challenges associated with the overall heterogeneity of HPC systems.

As evident with benchmark suites for heterogeneous systems, such as `Rodinia` [18], often multiple implementations are available for the same applications targeting CPU, GPU, and FPGA. Consequently, the strategic selection of the appropriate platform for job execution becomes crucial in achieving further performance improvements. In this section, we explore the prospect of a *Coordinated Approach* to HPC resource management.

Figure 6.1 illustrates a potential approach to such a coordinated approach to Heterogeneous HPC resource management. Here, we target node-level resource management for a heterogeneous system. Considering a global job queue $\mathcal{Q}$ and window size $\mathcal{W}$, we firstly need to classify the encountered jobs into platform-specific queues: $\mathcal{Q}_{CPU}$, $\mathcal{Q}_{GPU}$ and $\mathcal{Q}_{FPGA}$ for CPU, GPU, and FPGA, respectively. Next, we need to define platform-specific scheduling parameters: $\mathcal{W}_k$ and $\mathcal{C}^k_{max}$, where k can be CPU, GPU, or FPGA. Following the discussed mathematical formulation from Section3.1, we can again define the optimization problem at the platform-level as the strip packing problem [45]. The objective is to generate an optimal list of local queues (L_Q), and a list of complementary job sets (JS^k_i) along with corresponding resource allocations (R^k_i) per platform. Note



Global Job Queue : $\mathcal{Q}$
Window Size : $\mathcal{W}$
J_9 J_8 J_7 J_6 J_5 J_4 J_3 J_2 J_1
Platform Selection (Classification)
$\mathcal{W}_{CPU}$
J_6 J_4 J_1
Queue : $\mathcal{Q}_{CPU}$
$\mathcal{W}_{GPU}$
J_8 J_5 J_2
Queue : $\mathcal{Q}_{GPU}$
$\mathcal{W}_{FPGA}$
J_7 J_3
Queue : $\mathcal{Q}_{FPGA}$
CPU
GPU
FPGA
$\mathcal{C}_i^{CPU} \leq \mathcal{C}_{max}^{CPU}$
$\mathcal{C}_i^{GPU} \leq \mathcal{C}_{max}^{GPU}$
$\mathcal{C}_i^{FPGA} \leq \mathcal{C}_{max}^{FPGA}$
Time
JS_i^{CPU}
J_1 J_4 J_6
r_1 r_4 r_6
R_T^{CPU}
Resource Allocation
Time
JS_i^{GPU}
J_5 r_5 J_2 J_8 r_2 r_8
R_T^{GPU}
Resource Allocation
Time
JS_i^{FPGA}
J_5 r_5 J_3 r_3
R_T^{FPGA}
Resource Allocation
Local Queues $\rightarrow L_{\mathcal{Q}} = \{\mathcal{Q}_{CPU} : (J_1, J_4, J_6); \mathcal{Q}_{GPU} : (J_2, J_5, J_8); \mathcal{Q}_{FPGA} : (J_3, J_7)\}$
Co-scheduling $\rightarrow L_{JS}^{CPU}, L_{JS}^{GPU}, L_{JS}^{FPGA}$
Resource Allocation $\rightarrow L_R^{CPU}, L_R^{GPU}, L_R^{FPGA}$

Figure 6.1: Coordinated Approach to Heterogeneous HPC Resource Management

that k denotes the platform name. For throughput-oriented optimization, the main objective will be to minimize the overall execution time of the global job queue. In such a scenario, the set of resource management decisions can be performed at the *global* and *local* level.

1. **Global Decision (Platform Selection):** As illustrated in Figure 6.1, such a coordinated approach should classify jobs based on their platform-specific characteristics. Hence, a classification model needs to be built while considering job performance counters and hardware characteristics into account. There are certain existing works that try to model performance for heterogeneous systems such as [80, 47]. A further extension of these works can be a good starting point for such a classifier. If required, more sophisticated approaches using machine learning can also be explored. The global decisions shall yield a list of local queues L_Q, containing local queues for each platform.

2. **Local Decisions (Platform Specific):** The resource management problem at the platform level shall look very similar to our optimization problem (Section 3.1). For a given platform k, we need to formulate and solve the problem separately, as highlighted in this work. Our reinforcement learning-based approach can be useful to train platform-level agents. As each platform can have a very different set of resource allocation features and hardware configurations, it can be very challenging to train a single agent to make decisions for all platforms. Hence, it is vital to employ a platform-specific solution that can produce the list of job sets L_{JS^k} along with corresponding resource allocations L_R^k. Also, platform-specific scheduling can take place once a local queue has at least $\mathcal{W}_k$ jobs. Furthermore, platform-specific co-scheduling is initiated when a local queue accumulates a minimum of $\mathcal{W}_k$ jobs. Depending on the circumstances, the system can opt for additional waiting time to fill the queue or utilize time-shared scheduling. This is shown in Figure 6.1 for FPGA.

Furthermore, such an approach can be extended to a cluster scale. To this end, several improvements to the global and local decision making might be required. For implementing such extension, further integration can also be done with an existing cluster management tool such as Slurm [76].

6.3 Comparison with Existing Works

Finally, in this section, we compare our work with other existing methodologies. Following the literature survey conducted in Section 2.5, we have chosen five existing methods based on the proximity of their focus to our research. Table 6.1 presents the comparison of these methods concerning key ideas, methodology, and main distinctions from our work. Notable distinctions between our work and existing related works are also highlighted below:

1. Q. Zhu et al. [97] proposed pioneering work on CPU-GPU Heterogeneous systems, focusing on platform selection for portable workloads. However, their approach is limited to concurrently scheduling a single job on the CPU and GPU, respectively.

2. E. Arima et al. [4] offers a comprehensive analysis of the NVIDIA MIG feature, presenting a linear regression-based solution for throughput and fairness-oriented optimizations. While foundational for understanding NVIDIA MIG's impact, it does not leverage higher GPU concurrency and does not target CPUs.

3. I. Saba et al. [71] coordinates HPC resource management by addressing co-scheduling, resource partitioning, and power budgeting for heterogeneous systems. While similar to our scenario, their approach lacks scalability with respect to concurrency and lacks provisions for isolating shared resources on CPUs.

4. D. Zhang et al. [95] focuses on HPC Batch Job Scheduling, optimizing for metrics such as average job waiting time, average response time, average slowdown, and resource utilization using a reinforcement learning-based solution. While their solution methodology is similar to ours, the target scenario is quite different.

5. R. Chen et al. [20] has methodology proximity, but their approach has been designed for commodity hardware. Also, it differs considerably from our approach as it lacks NUMA-awareness and consideration of GPU-based systems. The dynamic resource partitioning in their approach motivates us (as discussed in Section 6.1.3) to extend our approach to include dynamic resource allocation decisions.

Additionally, Table 6.2 compares these methods based on available target scenarios. The parameters in Table 6.2 are considered based on our distinct use case, confirming the novelty and versatility of our work. As shown in this Table, our approach targets all scenarios except power capping. Power capping can be incorporated into our approach by adjusting the reward functions accordingly. Note that the symbols used in Table 6.2 are defined as follows: (1) "✓" indicates that the target scenario is available, (2) "×" indicates that the target scenario is not available, and (3) "-" indicates that the target scenario is not applicable.

Table 6.1: Comparison with Existing Works for Methodology

Author(s)	**Key Idea(s)**	**Methodology**	**Main Distinctions**
Q. Zhu et al. [97]	Proposes co-scheduling under power caps using performance bounds for co-schedules	Uses heuristic-based job characterization with a *Greedy* scheduling approach	**(1)** Limited concurrency ($\mathcal{C}_{CPU} = \mathcal{C}_{GPU} = 1$); **(2)** Includes power capping
E. Arima et al. [4]	Co-scheduling using *MIG* feature for GPU Partitioning under Power Caps	Proposes a *regression-based* performance model for estimating co-scheduling throughput and fairness	**(1)** Limited concurrency ($\mathcal{C} = 2$); **(2)** Explores MIG-only setups for GPU
I. Saba et al. [71]	Co-optimize co-scheduling, resource partitioning, and power capping on CPU-GPU systems	Utilizes *neural network*-based performance modeling with a perfect matching graph-based scheduling algorithm	**(1)** Limited concurrency ($\mathcal{C} = 2$); **(2)** No shared resource partitioning for CPUs; **(3)** Explores MIG-only setups for GPU
D. Zhang et al. [95]	Targets HPC batch job scheduling	Proposes an *RL-based* solution for optimizing batch scheduling parameters such as wait time and response time	Utilizes RL for an entirely different scenario (HPC Batch Job Scheduling)
R. Chen et al. [20]	Dynamic resource partitioning for co-located jobs on commodity servers	Utilizes *RL* and *neural network*-based performance modeling for generating co-location and resource isolation decisions	**(1)** Targets commodity hardware; **(2)** Dynamic resource partitioning;

Table 6.2: Comparison with Existing Works for Target Scenarios

		CPU			GPU			
Author(s)	**Co-Run**	**NUMA-aware**	**Cache Isolation**	**Memory BW Isolation**	**Physical GPU Isolation**	**Logical GPU Isolation**	**Power Capping**	**Targets HPC Jobs**
Q. Zhu el at. [97]	✓	×	×	×	×	×	✓	✓
E. Arima et al. [4]	✓	-	-	-	✓	×	✓	✓
I. Saba et al. [71]	✓	×	×	×	✓	×	✓	✓
D. Zhang et al. [95]	×	×	×	×	-	-	×	✓
R. Chen et al. [20]	✓	×	✓	✓	-	-	×	×
Our Work	✓	✓	✓	✓	✓	✓	×	✓

7 Conclusion

In this work, we focused on *co-scheduling* and *resource partitioning* in modern NUMA systems and GPUs equipped with emerging resource partitioning features. Initially, we introduced and formulated the resource management challenge as a throughput-oriented optimization problem. Later, we propose our reinforcement learning-based solution, aiming to discover the optimal set of decisions for co-scheduling and resource partitioning on a platform-level.

For NUMA systems, we explored the co-relation between NUMA-aware core/memory assignments and emerging hardware partitioning features, such as Intel CAT and Intel MBA. Additionally, we investigated the impact of job pair selections. Based on these observations, we implemented our solution for NUMA systems using the Soft Actor-Critic method, achieving a substantial throughput improvement of up to 78.1%.

With respect to GPUs, we focused on hierarchial resource management using partitioning features found in contemporary commercial GPUs, including MPS and MIG. We examined the benefits of these resource partitioning features along with the impact of hierarchical resource allocations. We proposed a deep-Q-learning based approach to co-optimize the configurations of these multiple and hierarchical resource partitioning features, along with making co-scheduling decisions for a given set of jobs. The experimental results showed the effectiveness of our approach, showcasing a significant throughput improvement of up to 87.3% for GPUs.

There are numerous opportunities for extending our work. One promising direction is the expansion of our work to include multiple nodes on a broader cluster scale. To achieve this, updates to the agent and the reward function are needed, potentially involving the integration of a larger and more scalable neural network or the introduction of multi-level entities to handle the increased complexity in such a setting. Additionally, there are prospects of integrating our approach with established HPC cluster management tools, like Slurm. Further, there is room for exploration into additional partitioning features spanning various components. The inclusion of other resources, such as power, can further enhance the comprehensiveness of our optimization framework. We also provided a high-level overview of a coordinated approach for resource management in HPC systems, suggesting a potential approach targeting heterogeneity in HPC systems. Lastly, we provide a comprehensive comparison of our approach to other existing works.

Scientific Contributions

The content of this thesis has been organized into two research papers, one of which has been presented at a peer-reviewed scientific conference. The details are provided as follows:

1. Urvij Saroliya, Eishi Arima, Dai Liu, and Martin Schulz "Hierarchical Resource Partitioning on Modern GPUs: A Reinforcement Learning Approach" In Proceedings of IEEE International Conference on Cluster Computing (CLUSTER), pp.185-196, November (2023) [72].
2. Urvij Saroliya, Eishi Arima, Dai Liu, and Martin Schulz "Harmonized Co-scheduling and Diverse Resource Assignments on NUMA Systems through Reinforcement Learning" [Under Review].

Scientific Engagement at IEEE Cluster 2023: As part of the research conducted for this master thesis, IEEE International Conference on Cluster Computing (CLUSTER) was attended in Santa Fe, New Mexico (USA). The conference served as an exceptional opportunity to present a segment of this thesis focused on GPUs (as described in chapter 5) and engage in exchange of scientific ideas within the field.

The IEEE Cluster serves as a major international forum for presenting and sharing recent accomplishments and technological developments in the field of cluster computing as well as the use of cluster systems for scientific and commercial applications [29]. The Cluster 2023 targeted discussions on recent advances in cluster computing, specifically focusing on (1) Applications, Algorithms, and Libraries; (2) Architecture, Networks/Communication, and Management; (3) Programming and Systems Software; and (4) Data, Storage, and Visualization [29].

Our work titled, "Hierarchical Resource Partitioning on Modern GPUs: A Reinforcement Learning Approach", was submitted to the *Architecture, Networks/Communication, and Management* submission track. Following a rigorous review and rebuttal process, the paper was accepted for presentation, with an overall acceptance rate of 23.84% for IEEE Cluster 2023 [91].

The conference presented a unique opportunity to participate in technical sessions covering diverse topics such as Distributed Machine Learning, Resource Management for HPC Systems, ML for Scheduling and Management, and GPU and FPGA Applications.

Noteworthy keynote addresses included discussions on "AI, Cloud, and the Future of HPC," "Pushing RISC-V into HPC," and an "Update on the Aurora Supercomputer."

On the sidelines of the conference, discussions were also conducted on future opportunities for extending the work presented in this thesis. There was considerable interest in exploring the latest resource partitioning features available on modern GPUs. Discussions were conducted around the potential of similar evaluations (as presented in Chapter 5) for deep learning workloads on GPUs. Additionally, integration possibilities with state-of-the-art cluster management tools, such as *Slurm* [76], were explored. Overall, there was significant anticipation for the future outcomes of the REGALE Project [68].

Ongoing Publication Efforts: Ongoing efforts are in place to publish the segment of this thesis focused on NUMA Systems (as described in chapter 4) with title, "Harmonized Co-scheduling and Diverse Resource Assignments on NUMA Systems through Reinforcement Learning", in a peer-reviewed scientific conference. Note that this research paper is currently undergoing the review process.

Abbreviations

AI	Artificial Intelligence
AM	Arithmetic Mean
API	Application Programming Interface
App	Application
BW	Bandwidth
CAT	Cache Allocation Technology
CBM	Capacity BitMask
CC-NUMA	Cache-Coherent Non-Uniform Memory Access
CI	Compute Instance
COS	Class of Service
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FCFS	First Come First Serve
FIFO	First In First Out
FP64	Float64 (Double-Precision Floating-Point Format)
FPGA	Field Programmable Gate Array
GB	Gigabyte
GiB	Gibibyte
GI	GPU Instance
GPU	Graphics Processing Unit
GPC	Graphics Processing Cluster
HBM	High Bandwidth Memory
HPC	High Performance Computing

ID	Identification
LLC	Last-Level Cache
MBA	Memory Bandwidth Allocation
MDP	Markov Decision Process
MIG	Multi-Instance GPU
MPS	Multi-Process Service
NP-hard	Non-deterministic Polynomial-time hard
NUMA	Non-Uniform Memory Access
OCRP-ML	Orchestrated Co-Scheduling, Resource Partitioning, and Power Capping on CPU-GPU Heterogeneous Systems via Machine Learning
OS	Operating System
PCIe	Peripheral Component Interconnect Express
QLC	Quad-Level Cell
QoS	Quality of Service
RDT	Resource Director Technology
RISC	Reduced Instruction Set Computer
RL	Reinforcement Learning
SAC	Soft Actor-Critic
SM	Streaming Multi-processor
SSD	Solid State Drive
TDP	Thermal Design Power
UPI	Ultra Path Interconnect
VLSI	Very Large Scale Integration

List of Figures

List of Tables

Bibliography

[1] T. Allen, X. Feng, and R. Ge. “Slate: Enabling Workload-Aware Efficient Multiprocessing for Modern GPGPUs.” In: *IPDPS*. 2019, pp. 252–261.

[2] D. Álvarez et al. “nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling.” In: *arXiv preprint arXiv:2204.10768* (2022).

[3] G. M. Amdahl. “Computer architecture and Amdahl’s law.” In: *Computer* 46.12 (2013), pp. 38–46.

[4] E. Arima, M. Kang, I. Saba, J. Weidendorfer, C. Trinitis, and M. Schulz. “Optimizing Hardware Resource Partitioning and Job Allocations on Modern GPUs under Power Caps.” In: *ICPP Workshops*. 2022.

[5] G. Aupy et al. “Co-Scheduling HPC Workloads on Cache-Partitioned CMP Platforms.” In: *CLUSTER*. 2018, pp. 348–358.

[6] R. Ausavarungnirun, V. Miller, J. Landgraf, S. Ghose, J. Gandhi, A. Jog, C. J. Rossbach, and O. Mutlu. “MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency.” In: *ASPLOS*. 2018, pp. 503–518.

[7] D. H. Bailey et al. “The NAS Parallel Benchmarks—Summary and Preliminary Results.” In: *Supercomputing*. 1991, pp. 158–165.

[8] M. Bhadauria et al. “An Approach to Resource-aware Co-scheduling for CMPs.” In: *ICS*. 2010, pp. 189–199.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications.” In: *PACT*. 2008, pp. 72–81.

[10] J. Bircsak et al. “Extending OpenMP for NUMA machines.” In: *SC*. 2000, pp. 48–48.

[11] S. Blagodurov et al. “A case for NUMA-aware contention management on multicore systems.” In: *PACT*. 2010, pp. 557–558.

[12] W. Bolosky et al. “Simple but effective techniques for NUMA memory management.” In: *SOSP*. 1989, pp. 19–31.

[13] W. J. Bolosky et al. “NUMA Policies and Their Relation to Memory Architecture.” In: *ASPLOS*. 1991, pp. 212–221.

[14] J. Breitbart et al. “Case Study on Co-scheduling for HPC Applications.” In: *ICPPW*. 2015, pp. 277–285.

[15] J. Breitbart et al. “Dynamic Co-Scheduling Driven by Main Memory Bandwidth Utilization.” In: *CLUSTER*. 2017, pp. 400–409.

[16] A. D. Breslow, A. Tiwari, M. Schulz, L. Carrington, L. Tang, and J. Mars. “Enabling fair pricing on hpc systems with node sharing.” In: *SC*. 2013, pp. 1–12.

[17] A. Calotoiu, D. Beckinsale, C. W. Earl, T. Hoefler, I. Karlin, M. Schulz, and F. Wolf. “Fast Multi-parameter Performance Modeling.” In: *CLUSTER*. 2016, pp. 172–181.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. “Rodinia: A Benchmark Suite for Heterogeneous Computing.” In: *IISWC*. 2009, pp. 44–54.

[19] Q. Chen, H. Chung, Y. Son, Y. Kim, and H. Y. Yeom. “SmCompactor: A Workload-Aware Fine-Grained Resource Management Framework for GPGPUs.” In: *SAC*. 2021, pp. 1147–1155.

[20] R. Chen et al. “DRLPart: A Deep Reinforcement Learning Framework for Optimally Efficient and Robust Resource Partitioning on Commodity Servers.” In: *HPDC*. 2021, pp. 175–188.

[21] J. Choi et al. “Interference-aware co-scheduling method based on classification of application characteristics from hardware performance counter using data mining.” In: *Cluster Comp.* 23 (2020), pp. 57–69.

[22] P. Christodoulou. “Soft actor-critic for discrete action settings.” In: *arXiv preprint arXiv:1910.07207* (2019).

[23] B. Cumming. *STREAM Benchmark in CUDA C++*. `https://github.com/bcumming/cuda-stream`. Accessed: Apr 9, 2023. 2017.

[24] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou. “Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls.” In: *HPCA*. 2018, pp. 208–220.

[25] R. H. Dennard et al. “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions.” In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[26] N. Ding and S. Williams. “An Instruction Roofline Model for GPUs.” In: *PMBS*. 2019, pp. 7–18.

[27] F. Foundation. *Gymnasium Documentation*. `https://gymnasium.farama.org/`. Accessed: Apr 30, 2023. 2022.

[28] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor." In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[29] *IEEE CLUSTER 2023*. `https://clustercomp.org/2023/`. Accessed: Dec 10, 2023.

[30] S. Imamura et al. "Power-capped DVFS and thread allocation with ANN models on modern NUMA systems." In: *ICCD*. 2014, pp. 324–331.

[31] Intel. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3B. 2023. Chap. 18 Debug, Branch Profile, TSC, and Intel Resource Director Technology (Intel(R) RDT) Features, pp. 18.52–18.70.

[32] Intel. *Intel(R) RDT Software Package*. `https://github.com/intel/intel-cmt-cat`. Accessed: Nov 15, 2023.

[33] E. Ipek et al. "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach." In: *ISCA*. 2008, pp. 39–50.

[34] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. "An approach to performance prediction for parallel applications." In: *Euro-Par*. 2005, pp. 196–205.

[35] J. Jeffers, J. Reinders, and A. Sodani. *Intel Xeon Phi processor high performance programming: knights landing edition*. Morgan Kaufmann, 2016.

[36] *Kernel Probes (Kprobes)*. `https://docs.kernel.org/trace/kprobes.html`. Accessed: Nov 15, 2023.

[37] J. Kim, J. Kim, and Y. Park. "Navigator: Dynamic Multi-kernel Scheduling to Improve GPU Performance." In: *DAC*. 2020, pp. 1–6.

[38] A. Kleen. "A numa api for linux." In: *Novel Inc* (2005).

[39] A. Lai. *random-access-bench*. `https://github.com/cowsintuxedos/random-access-bench`. Accessed: Apr 9, 2023. 2018.

[40] B. Lepers et al. "Thread and Memory Placement on NUMA Systems: Asymmetry Matters." In: *USENIX ATC*. 2015, pp. 277–289.

[41] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari. "MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters." In: *SoCC*. 2022, pp. 173–189.

[42] H. Li et al. "Locality and loop scheduling on NUMA multiprocessors." In: *ICPP*. 1993, pp. 140–147.

[43] Linux. *NUMACTL(8) - linux man page*. `https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html`. Accessed: Nov 15, 2023.

[44] LLNL. *CORAL-2 Benchmarks*. https://asc.llnl.gov/coral-2-benchmarks. Accessed: Apr 9, 2023. 2017.

[45] S. Martello, M. Monaci, and D. Vigo. "An exact approach to the strip-packing problem." In: *INFORMS Journal on Computing* 15.3 (2003), pp. 310–319.

[46] C. McCurdy et al. "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms." In: *ISPASS*. 2010, pp. 87–96.

[47] J. C. Meyer and A. C. Elster. "Performance modeling of heterogeneous systems." In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE. 2010, pp. 1–4.

[48] V. Mnih et al. "Human-level control through deep reinforcement learning." In: *nature* 518.7540 (2015), pp. 529–533.

[49] O. Mutlu and T. Moscibroda. "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems." In: *ISCA*. 2008, pp. 63–74.

[50] M. A. Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015. Chap. 2.

[51] K. Nikas et al. "DICER: Diligent Cache Partitioning for Efficient Workload Consolidation." In: *ICPP*. 2019.

[52] Nvidia. *Multi-Process Service documentation*. https://docs.nvidia.com/deploy/mps/index.html. Accessed: Nov 15, 2023.

[53] Nvidia. *Nsight Compute CLI*. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html. Accessed: Nov 15, 2023.

[54] Nvidia. *NVIDIA A100 Tensor Core GPU Architecture*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. Accessed: Nov 30, 2023. 2020.

[55] Nvidia. *NVIDIA multi-instance GPU*. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. Accessed: Nov 15, 2023.

[56] Nvidia. *NVIDIA Nsight Compute*. https://developer.nvidia.com/nsight-compute. Accessed: Nov 15, 2023.

[57] *NVIDIA Ampere Architecture In-Depth*. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/. Accessed: Dec 10, 2023.

[58] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan. "Improving GPGPU Concurrency with Elastic Kernels." In: *ASPLOS*. 2013, pp. 407–418.

[59] J. Park et al. “CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers.” In: *EuroSys*. 2019.

[60] J. Park et al. “Hypart: A Hybrid Technique for Practical Memory Bandwidth Partitioning on Commodity Servers.” In: *PACT*. 2018.

[61] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. “PyTorch: An Imperative Style, High-Performance Deep Learning Library.” In: *NIPS*. 2019, pp. 8024–8035.

[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay. “Scikit-learn: Machine Learning in Python.” In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[63] *Perf Wiki*. `https://perf.wiki.kernel.org/`. Accessed: Nov 15, 2023.

[64] M. L. Puterman. “Chapter 8 markov decision processes.” In: *Handbooks in operations research and management science* 2 (1990), pp. 331–434.

[65] M. K. Qureshi et al. “Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches.” In: *MICRO*. 2006, pp. 423–432.

[66] N. Rafique et al. “Effective Management of DRAM Bandwidth in Multicore Processors.” In: *PACT*. 2007, pp. 245–258.

[67] C. Reano, F. Silla, D. S. Nikolopoulos, and B. Varghese. “Intra-Node Memory Safe GPU Co-Scheduling.” In: *IEEE TPDS* 29.5 (2018), pp. 1089–1102.

[68] *REGALE - Open Architecture for Exascale Supercomputers*. `https://regale-project.eu/`. Accessed: Dec 10, 2023.

[69] *Reinforcement Learning 101*. `https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292`. Accessed: Dec 10, 2023.

[70] I. S. Barrera et al. “Modeling and Optimizing NUMA Effects and Prefetching with Machine Learning.” In: *ICS*. 2020.

[71] I. Saba et al. “Orchestrated Co-Scheduling, Resource Partitioning, And Power Capping On CPU-GPU Heterogeneous Systems Via Machine Learning.” In: *ARCS*. 2022, pp. 51–67.

[72] U. Saroliya et al. "Hierarchical Resource Partitioning on Modern GPUs: A Reinforcement Learning Approach." In: *CLUSTER*. 2023, pp. 185–196.

[73] H. Sasaki et al. "Scalability-based Manycore Partitioning." In: *PACT*. 2012, pp. 107–116.

[74] V. S. da Silva et al. "Smart resource allocation of concurrent execution of parallel applications." In: *Concurrency and Computation: Practice and Experience* 35.17 (2023), e6600.

[75] G. Singh et al. "Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning." In: *ISCA*. 2022, pp. 320–336.

[76] *Slurm workload manager*. `https://slurm.schedmd.com/`. Accessed: Nov 30, 2023.

[77] P. Sohal, M. Bechtel, R. Mancuso, H. Yun, and O. Krieger. "A closer look at intel resource director technology (rdt)." In: *RTNS*. 2022, pp. 127–139.

[78] R. S. Sutton et al. *Reinforcement learning: An introduction*. MIT press, 2018.

[79] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. "Enabling Preemptive Multiprogramming on GPUs." In: *ISCA*. 2014, pp. 193–204.

[80] L. Tang, X. S. Hu, and R. F. Barrett. "PerDome: a performance model for heterogeneous computing systems." In: *Proceedings of the Symposium on High Performance Computing*. 2015, pp. 225–232.

[81] *The Universal Approximation Theorem*. `https://www.deep-mind.org/2023/03/26/the-universal-approximation-theorem/`. Accessed: Dec 8, 2023.

[82] TOP500. *TOP 500*. `https://www.top500.org/statistics/list/`. Accessed: Nov 15, 2023.

[83] *Tutorial: Linux kernel profiling with perf*. `https://perf.wiki.kernel.org/index.php/Tutorial`. Accessed: Nov 15, 2023.

[84] *Uprobe-tracer: Uprobe-based Event Tracing*. `https://docs.kernel.org/trace/uprobetracer.html`. Accessed: Nov 15, 2023.

[85] *Using the Linux Kernel Tracepoints*. `https://docs.kernel.org/trace/tracepoints.html`. Accessed: Nov 15, 2023.

[86] H. Van Hasselt, A. Guez, and D. Silver. "Deep reinforcement learning with double q-learning." In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[87] B. Verghese et al. "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers." In: *ASPLOS*. 1996, pp. 279–289.

[88] Y. Wang et al. "Online Power Management for Multi-Cores: A Reinforcement Learning Based Approach." In: *IEEE TPDS* 33.4 (2022), pp. 751–764.

[89] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas. "Dueling network architectures for deep reinforcement learning." In: *International conference on machine learning*. PMLR. 2016, pp. 1995–2003.

[90] C. J. C. H. Watkins. "Learning from delayed rewards." In: (1989).

[91] "Welcome Message from the IEEE Cluster 2023 Program Chairs." In: *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 2023, pp. 12–13.

[92] *What is a neural network?* `https://aws.amazon.com/what-is/neural-network/`. Accessed: Dec 8, 2023.

[93] Y. Xiang et al. "EMBA: Efficient Memory Bandwidth Allocation to Improve Performance on Intel Commodity Processor." In: *ICPP*. 2019.

[94] S. Yoo et al. "Reinforcement Learning-Based SLC Cache Technique for Enhancing SSD Write Performance." In: *HotStorage*. 2020.

[95] D. Zhang et al. "RLScheduler: An Automated HPC Batch Job Scheduler Using Reinforcement Learning." In: *SC*. 2020, pp. 1–15.

[96] P. Zhang et al. "ReSemble: Reinforced Ensemble Framework for Data Prefetching." In: *SC*. 2022.

[97] Q. Zhu et al. "Co-Run Scheduling with Power Cap on Integrated CPU-GPU Systems." In: *IPDPS*. 2017, pp. 967–977.

[98] P. Zou et al. "Contention aware workload and resource co-scheduling on power-bounded systems." In: *NAS*. 2019, pp. 1–8.