Future Generation Computer Systems 181 (2026) 108428

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Real-time AI-powered monitoring for energy-efficient scheduling in multi-node heterogeneous systems

Taha Abdelazziz Rahmani [a,b,*], Ghalem Belalem [b], Sidi Ahmed Mahmoudi [c], Omar Rafik Merad-Boudia [b]

[a] Computer Science department, University of Roma Sapienza, 295, Viale Regina Elena, Roma, 00161, Italy
[b] Computer Science department, LIO laboratory, University of Oran1, Ahmed Ben Bella, N2A, Es Senia, Oran, 31000, Algeria
[c] Computer Science department, Faculty of Engineering,University of Mons, 9, Rue de Houdain, Mons, 7000, Belgium

ARTICLE INFO

Keywords:
Execution time prediction
Heterogeneous computing
Deep learning
Machine learning
Load balancing
Energy

ABSTRACT

Load balancing is critical for maintaining computing systems' stability and achieving optimal performance. Its significance is widely recognized across different computing fields, particularly in the context of heterogeneous systems. These systems comprise computing devices with varying computational capabilities and architectures, each optimized for specific workloads. This heterogeneity introduces dynamic resource constraints, architectural mismatches, and unpredictable task-device affinity, which aggravates the challenges of load balancing. This paper presents an AI-driven load balancing solution for real-time distributed heterogeneous systems. Our approach continuously monitors the system state, capturing key factors that influence performance, such as task and device characteristics. Leveraging AI-based models, it computes a dynamic load index for each device based on the collected data. Using these load estimations, the method predicts potential imbalances through a novel imbalance metric and proactively schedules incoming applications to the most suitable devices, ensuring system-wide balance. To validate our approach, we first evaluated the prediction models by comparing a variety of machine learning algorithms with device-specific deep learning models, with the latter achieving superior accuracy. We then compared our method against widely used scheduling techniques across diverse workloads. The results show that our approach achieves more balanced workload distribution, faster execution, higher throughput, improved resource utilization, and reduced energy consumption across all scenarios, showcasing its adaptability to dynamic conditions and its applicability in real-world settings.

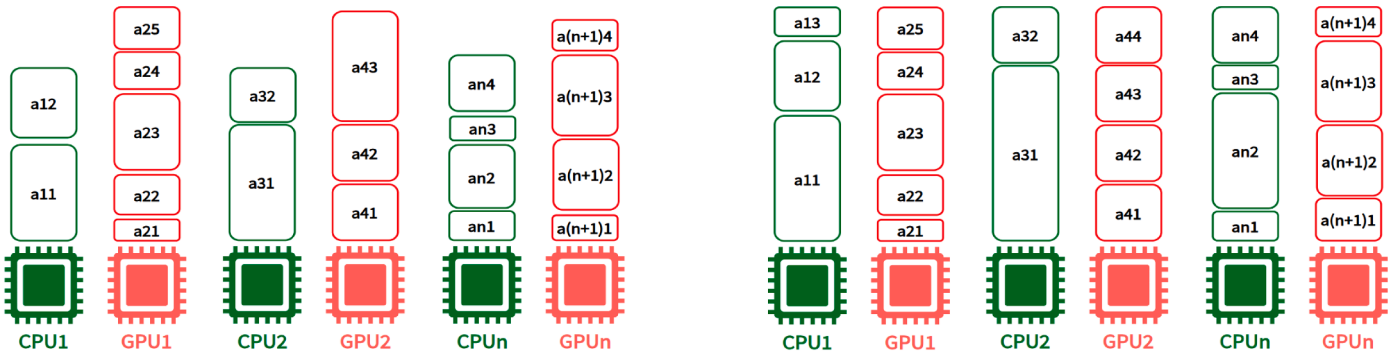
1. Introduction

The rapid development of data-intensive applications, such as artificial intelligence (AI) and the internet of things (IoT), has generated unprecedented volumes of data, pushing traditional CPU-only systems beyond their limits. To meet these demands, modern computing platforms have integrated more powerful accelerators, most notably Graphics Processing Units (GPUs). This integration has transformed formerly homogeneous CPU-based architectures into heterogeneous systems, where CPUs and GPUs operate across multiple nodes. In these CPU-GPU heterogeneous systems, GPUs act as co-processors to CPUs, leveraging their complementary strengths: GPUs excel at data-parallel, compute-intensive tasks, while CPUs efficiently handle sequential and latency-sensitive operations.

While heterogeneous computing enhances flexibility by efficiently executing diverse workloads, it introduces significant challenges due to architectural differences and unpredictable task-device affinities. Relying solely on application-device suitability often results in overloading powerful devices while leaving others underutilized. To illustrate this issue, Fig. 1 conceptually depicts two workload distribution scenarios in a heterogeneous cluster executing a mix of CPU-suitable and GPU-suitable applications. In the first scenario, a scheduler that considers only device suitability tends to assign most compute-intensive tasks to the GPU, as shown in Fig. 1a. Consequently, the GPU's queue becomes overloaded with demanding tasks requiring extensive computation time, while the CPUs, having completed their lighter workloads earlier, remain idle. This imbalance, caused by disregarding current load conditions and workload characteristics, leads to longer execution times, lower throughput, and higher energy consumption. In contrast, Fig. 1b illustrates a balanced state achieved by considering both device suitability and current load when assigning tasks. By offloading part of the GPU's workload to idle CPUs, the scheduler achieves greater device utilization, reduced response times, higher throughput, and improved energy efficiency.

* Corresponding author.
E-mail address: rahmani@di.uniroma1.it (T.A. Rahmani).

https://doi.org/10.1016/j.future.2026.108428
Received 26 June 2025; Received in revised form 5 February 2026; Accepted 10 February 2026
Available online 13 February 2026



(a) Imbalanced workload distribution

(b) Balanced workload distribution

Fig. 1. Balanced vs. imbalanced load in a CPU-GPU cluster.

Effective scheduling in heterogeneous systems requires evenly distributing workloads across devices. Achieving this load balance in real time is particularly challenging, as it requires addressing the dynamic nature of workloads and the diverse capabilities of hardware components during application scheduling. The lack of built-in scheduling mechanisms in existing programming frameworks further complicates the process, forcing developers to manage synchronization manually at a low level. As a result, traditional strategies originally designed for homogeneous systems prove inadequate, highlighting the need for context-aware schedulers that can adapt to dynamic and complex execution conditions.

Continuous monitoring of device load is crucial for effective load balancing in heterogeneous systems. It allows schedulers to proactively prevent overloads by optimizing workload distribution at the time of task assignment, thereby eliminating the need for application migration. In contrast, reactive strategies such as work-stealing schedulers [1] rely on migrating tasks from overloaded to underloaded devices, introducing substantial overhead in execution time and bandwidth consumption [2].

However, accurate monitoring requires a precise, context-aware definition of device load that captures the interplay between the device's capabilities and application's requirements. Architectural features such as memory hierarchy determine the device's computational capability, while application characteristics like arithmetic intensity define workload demands. Any changes in these factors must be dynamically captured in load measurements; otherwise, the scheduler risks inaccurate assessments that lead to inefficient workload distribution and suboptimal performance.

Prior research across various domains typically defines device load using metrics such as total processed data bytes [3] or CPU/memory utilization [4]. However, these metrics focus exclusively on currently executing applications while neglecting queued workloads. Pending applications still occupy positions in execution pipelines and will consume processing resources once initiated, thereby delaying subsequent tasks. Ignoring queued workload provides an incomplete assessment of device utilization, potentially leading schedulers to overload already burdened devices. Consequently, such metrics, although useful for performance assessment, are inadequate for real-time scheduling in heterogeneous systems [5].

To address this problem, we define load as the total time required to execute all applications currently pending on a given device. This definition provides a precise measure of load, as execution time reflects both device and application characteristics. However, this approach requires prior knowledge of execution times, which is unavailable at submission time [6].

To overcome this limitation, we predict the execution time of applications to calculate device loads. Execution time prediction can be approached using two methods: traditional methods and artificial intelligence methods. Traditional methods rely on manual calculations, which tend to produce less accurate and reliable results [7]. In contrast, artificial intelligence predictors have demonstrated strong effectiveness, even under highly dynamic workloads and at larger scales, such as workload forecasting in cloud data centers [8,9]. AI approaches can capture the complex interrelationships between application and device characteristics, achieving higher prediction accuracy.

Nevertheless, monitoring individual device load only provides local state information without capturing system-wide load distribution. Evaluating the whole load distribution requires a metric that quantifies the disparity in load across all available devices, representing the overall system imbalance. Previous works have defined imbalance primarily as a performance evaluation metric rather than a monitoring tool. For instance, the degree of imbalance (DI) presented in work [10] measures the deviation between the most loaded and least loaded devices from the average, focusing on worst-case scenarios to prevent extreme overload or underload conditions.

In this work, we define imbalance as the cumulative deviation of all devices from the average load. This definition enables the scheduler to capture the complete workload distribution and actively minimize system imbalance when assigning new tasks. Unlike DI that focuses on extremes, our metric penalizes small load disparities that are distributed across the system, allowing finer-grained optimization. Such imbalances may be overlooked by DI when maximum and minimum loads are close, while many devices still deviate moderately from the mean.

This paper introduces Smart-Balance, a novel AI-driven scheduling framework that proactively mitigates load imbalances in real-time multi-node CPU-GPU heterogeneous systems. Smart-Balance employs AI models to predict the load, compute system imbalance, and assign each application to the device that minimizes this metric, ensuring an equitable workload distribution across the system.

In this study, our key contributions include:

- A comparative evaluation of multiple machine learning (ML) models with device-specific deep learning (DL) models for predicting application execution times.
- The integration of AI techniques into real-time heterogeneous systems, addressing challenges in application scheduling and load monitoring.
- Identification and emperical validation of pivotal factors influencing load balancing in heterogeneous systems.
- Introduction of a novel imbalance metric for load monitoring.
- Adaptation of prevalent load balancing techniques from diverse domains to validate our approach.

The subsequent sections of the paper are structured as follows: Section 2 provides background information on heterogeneous systems, load balancing, and an overview of related research. Section 3 introduces our proposed load balancing scheduler. Section 4 discusses the construction of devices' datasets and the training of the execution time prediction

models. Section 5 details the results of the conducted experiments. Finally, Section 6 provides a summary of our work and discusses potential improvements for future research.

2. Background

The increasing adoption of GPUs in various computing systems has led to a growing research interest in CPU-GPU heterogeneous systems. This shift is attributed to the accessibility and simplified programmability of GPUs, which offer substantial advantages for general-purpose computing when used concurrently with CPUs.

2.1. Programming heterogeneous systems

In heterogeneous systems, applications are typically developed following the host-kernel model [11]. In this model, the kernel, representing the parallel component of the application, executes computational operations and can run on various devices, including CPUs and GPUs. The host, on the other hand, manages the sequential aspects of the application, coordinating kernel execution across devices, and runs exclusively on the CPU.

OpenCL [12] is a widely adopted framework for programming heterogeneous systems. In the OpenCL paradigm, kernel code is executed by launching multiple threads, known as work-items, each representing an instance of the kernel. These work-items are grouped into work-groups, with each work-group executing on a single computing unit. OpenCL allows work-groups of a single kernel to run independently, and different work-groups can execute on different devices.

Scheduling applications in heterogeneous systems can be approached via two methods: data-parallel and task-parallel [13]. Data-parallel approaches enable the co-execution of a single application's kernel across multiple devices, while task-parallel approaches assign each kernel to a single device. Selecting the appropriate scheduling approach is crucial and depends on both the computational demands of the applications and the capabilities of the devices within the system [5].

Executing applications with large data volumes on individual devices can lead to performance bottlenecks due to memory and processing constraints. In such cases, employing a data-parallel approach is essential, as it shares the workload across multiple devices, reducing the computational strain on any single device. However, data-parallel methods are more complex to implement, as they require additional mechanisms for optimal data partitioning based on each device's capabilities. Moreover, the overall speedup must justify the overhead incurred during data partitioning and transfer between devices [2].

In contrast, task-parallel approaches are simpler to implement and incur minimal transfer overhead, as data are transferred only at the beginning and end of each application's execution. However, these methods are suitable for applications with smaller data volumes than data-parallel approaches [2].

2.2. Load balancing

Load balancing ensures a balanced distribution of workloads across computing resources to optimize system performance. It can be classified into two categories: static and dynamic [14].

Static load balancing methods assign tasks based on a predefined resource allocation derived from prior system knowledge, such as job submission times and resource requirements, without accounting for real-time resource states. Although simple to implement, these methods lack the flexibility required to adapt to variations in workloads and system states.

In contrast, dynamic load balancing algorithms are more complex but offer greater efficiency. They continuously adapt to the system's evolving state by making real-time adjustments during scheduling. Although they may incur additional overhead, their ability to adapt makes them particularly well-suited for managing dynamic and unpredictable workloads.

2.3. Related works

In this section, we categorize previous studies on heterogeneous systems based on the scheduling approach they adopt (data-parallel or task-parallel) and their primary objective (optimizing performance or load balance). We also incorporate relevant research from other domains, such as web services, to explore potential solutions that can be adapted to address the load balancing challenges in heterogeneous systems.

2.3.1. Data-parallel approaches

Performance Optimization. The authors of the work [15] implement the Principal Component Analysis (PCA) algorithm on a heterogeneous mobile system by partitioning it into two components: a high-precision and a computationally intensive segment. The high-precision component is executed on the CPU, while the GPU executes the computationally demanding instructions, effectively leveraging both processors where they excel.

The authors of the work [16] propose a dynamic scheduling approach for heterogeneous systems. This strategy selects the most suitable device for each sub-task of a job by considering multiple factors, including data transfer time, estimated execution time based on historical data, and the current workload on each device. Tasks are assigned to the processing unit that is predicted to complete them most quickly.

Load Balance. The authors of the work [17] explored different load balancing techniques for data-parallel applications in heterogeneous systems, including static, dynamic, and H-guided approaches.

The static load balancing algorithm partitions the overall workload into packages that match the number of devices within the system. Each device receives a package proportional to the amount of work it can execute within a given time unit. Conversely, the dynamic algorithm segments the total workload into a higher number of small, uniform-sized packages. Initially, each device executes a single package. Upon completing the execution of its designated package, the device proceeds to execute the subsequent packages in line. In cases where a device remains idle and no further packages are available in line, it steals packages from devices with excessive workloads. On the other hand, the H-guided load balancing method employs the same algorithm as the dynamic method for distributing packages but a different approach to determine package sizes. Unlike the dynamic approach, the H-guided algorithm continuously decreases the size of packages.

The authors of the work [18] present a scheduling method for optimizing single-node heterogeneous computing systems. They combined AI planning heuristics with machine learning to determine near-optimal system configuration, including the number of CPU threads, GPU threads, and the workload distribution between the CPU and GPU. AI planning heuristics guide the search towards configurations that offer the best trade-off between high performance and low energy consumption, intelligently navigating the parameter space to significantly reduce the number of configurations that need to be evaluated. Concurrently, a machine learning model estimates the performance per Joule of the selected configurations, allowing for rapid evaluation and effective decision-making.

PDAWL (Profile-based AI-assisted Dynamic Scheduling) [19] is a hybrid scheduling approach for single-node heterogeneous systems. This approach integrates several components to optimize workload distribution, combining online scheduling, application profiling, mathematical modeling, and offline machine learning prediction. The PDAWL scheduler dynamically adjusts workloads based on profiled application execution data and historical performance records. It employs a mathematical model to predict execution times for various workloads using these historical records, offering a coarse-grain selection of computing

resources. Additionally, machine learning models, trained with historical performance data, are used to predict execution and data transfer times more accurately. By combining these predictions, the scheduler assigns tasks to the most suitable computing resources, enhancing system performance.

2.3.2. Task-parallel approaches

Performance Optimization. KubeSCRTP [20] is a machine learning-based scheduler for the Kubernetes framework that minimizes the execution time of applications. It dynamically classifies applications using a pre-trained ML model into two categories: fast-executing and slow-executing. Fast-executing applications are assigned to the CPU, while slow-executing applications are offloaded to the GPU.

The study in reference [21] proposes a scheduling strategy for OpenCL programs in heterogeneous systems. This strategy predicts the most suitable device for each application by considering both its program attributes, such as instruction count, and the hardware's configuration. Subsequently, the scheduler assigns each application to its predicted device.

In reference [11], the authors introduce a scheduler that optimizes the execution of applications in CPU/GPU heterogeneous systems. The scheduler dynamically assigns applications to computing devices based on the speedup achieved when running applications on GPUs compared to CPUs. Leveraging machine learning to predict the speedup, the scheduler assigns applications to GPUs if the predicted speedup exceeds "one"; otherwise, the application is allocated to the CPU.

InEPS [22] is an AI-driven, energy-aware scheduling framework for heterogeneous HPC clusters. It uses key job metrics (e.g. submission time and requested cores) and node characteristics (e.g. power consumption, availability, and clock rate) as input features for a deep neural network to filter out infeasible job-node pairings (e.g., when a job exceeds a node's capacity). Subsequently, the InEPS approach applies deep reinforcement learning to select the best scheduling configuration, minimizing jobs' energy consumption.

DYPE [23] is a dynamic scheduling framework for irregular workloads in heterogeneous systems. It uses multiple machine learning models to predict kernel execution time, communication cost, and energy consumption. Subsequently, the approach evaluates all scheduling configurations and selects the one that best balances performance and energy consumption.

Load Balance. "Troodon" [24] scheduler classifies applications into CPU-suitable and GPU-suitable pools using a machine learning classifier. The applications are then sorted in each pool according to their predicted speedup, which is estimated using a regression machine learning model. The CPU-suitable pool is sorted in ascending order of speedup, while the GPU-suitable pool is sorted in descending order. The two pools are then combined, with the CPU-suitable pool at the top. Applications at the top of the pool are allocated to the CPU, while those at the bottom are allocated to the GPU. "Troodon" allocates applications to devices until the estimated load of each device is met. If a device cannot achieve its estimated load, it allocates non-suitable applications to that device.

In the work [25], the authors introduced a device suitability classifier for single node heterogeneous systems with multiple CPUs and GPUs. This classifier uses a machine learning model to predict the fastest device to run each OpenCL application. Subsequently, the scheduler executes each application on its predicted faster device.

The work in [1] proposes a work-stealing scheduler for single-node heterogeneous CPU-GPU systems. The scheduler employs a machine learning-based predictor to estimate the execution time of applications on both CPU and GPU, assigning tasks to the most suitable device. The system then dynamically redistributes tasks at runtime, stealing tasks from the queue of an overloaded device, and assigning it to the idle one.

In [26], the authors introduce "RTLB_Sched", a machine learning-driven load balancer for real-time CPU-GPU heterogeneous systems. "RTLB_Sched" optimizes resource allocation by dynamically estimating device loads using ML models and assigning applications to prevent workload imbalances, ensuring a balanced and efficient system.

In [27], the authors examine the impact of efficient load balancing on system performance. They introduce "HBalancer", a machine learning-driven scheduler that predicts and mitigates load imbalances by selecting the optimal scheduling configuration. By achieving better load distribution, "HBalancer" reduces application response times.

The study in [6] conducts a comparative analysis of common load balancing strategies in the domain of cluster-based web servers. These strategies include Round Robin, Weighted Round Robin, and Least Connections. The primary goal of this investigation is to assess the efficiency of these methods in distributing user requests across different server nodes.

In [28], the authors propose a novel scheduling technique called Priority-Weighted Round Robin (PWRR) for healthcare monitoring systems. PWRR assigns priorities to data to ensure that critical information is transmitted first. It uses weighted round robin (WRR) scheduling to allocate available bandwidth fairly among different devices. In WRR, each device is given a certain amount of time to transmit data. The device with higher weight gets more bandwidth. If the device is unable to transmit all its data during its allocated time, the remaining data is queued up and transmitted in the next round.

In the study [29], the authors provide a comprehensive evaluation of the Weighted Round Robin load balancing approach for cloud web services. They assign different weights to servers according to their specifications, such as CPU speed and memory. The study demonstrates the impact of correct weight assignments on overall system performance.

The paper [30] presents an experimental setup for popular load balancing algorithms such as Round Robin and Shortest Job First in a virtual cloud environment. The main objective of this investigation is to compare the performance of these algorithms in a simulated cloud environment.

In [31], the authors implement a Weighted Least Connection (WLC) scheduling algorithm in web cluster systems. The proposed algorithm assigns higher weights to servers with fewer connections, which improves load balancing by ensuring that servers with fewer connections are more likely to be selected for new requests. It also addresses the shortcoming of the traditional Least Connections algorithm of overloading servers by temporarily excluding them from the scheduling list.

2.3.3. Discussion

The studies presented in this section investigated different approaches to task scheduling in heterogeneous systems. Data-parallel scheduling strategies distribute an application's work-groups across multiple devices. The overhead associated with partitioning the kernel and transferring data between devices can be significant. Thus, the feasibility of data-parallel methods depends on the application having sufficiently high computational requirements to offset these overheads.

In contrast, task-parallel approaches are particularly suited for applications with lower computational intensity. These methods allow the system to accommodate a greater number of user applications, since each device can be assigned a different kernel, rather than limiting the entire system to execute a single kernel at a time. This approach is also advantageous for applications with irregular data dependencies, as devices can process their own data independently.

The works [11],[20],[21], and [25] primarily focus on minimizing applications' execution times. On the other hand, the works [22] and [23] assign tasks to devices that minimize energy consumption, trading energy efficiency against performance. However, these approaches typically prioritize individual task optimization over system-wide efficiency, often overlooking the current workload distribution. As a result, certain devices that are preferentially allocated tasks may become overloaded, leading to longer queueing times, increased response times, and higher overall energy consumption.

In contrast, the scheduler presented in [24] groups all applications into a single job pool, disregarding their submission timing and order. This approach assumes that the processing requirements of each application are known in advance and classifies them as either CPU- or GPU-suitable. However, this classification oversimplifies device differences by relying solely on computational power (measured in FLOPS), ignoring manufacturer-specific characteristics.

Real-time systems must contend with dynamic and unpredictable submissions. Since applications arrive at irregular intervals, predicting the computational requirements of all tasks in advance is infeasible. Thus, solutions that rely on precomputed workload data are impractical in these environments.

The studies in [6],[28],[29],[30], and [31] investigate various load balancing strategies. Among these, [28],[29], and [30] employ static approaches, such as Round-Robin and Weighted Round-Robin, to distribute applications. While simple, these static methods lack adaptability to changing workloads. In contrast, [6] and [31] use dynamic methods, including Least Connections and Weighted Least Connections, to distribute requests more fairly. However, these methods often overlook the distinct characteristics of individual requests.

The load balancing approaches proposed in [26],[27], and [1] address many of these issues by considering all key features to achieve an equitable load distribution across devices in a heterogeneous system. Nevertheless, these methods were designed for single-node CPU-GPU systems and cannot be directly deployed to more complex configurations.

In summary, the limitations of the discussed approaches are:

- Code splitting: challenging to implement and introduces additional time overhead.
- Ignoring load balance.
- Lack of system state monitoring.
- Sequential application submission into a job pool.
- Prior knowledge requirements.
- Oversimplification of heterogeneity, neglecting differences in device capabilities and application requirements.

In the following section, we discuss our solution that addresses these limitations and leverages the strengths of heterogeneous systems to achieve enhanced system performances.

3. Smart-balance

In this section, we introduce our proposed scheduler, Smart-Balance. We provide details on the system architecture, mathematical model, and the scheduling algorithm.

3.1. System architecture

In real-time heterogeneous systems, applications are continuously submitted for execution. The scheduler must promptly assign them to computing devices to minimize overhead.

Each submitted application has a number of scheduling options that is equal to the number of available devices. For every option, the scheduler predicts the resulting system imbalance before any actual assignment (see Section 3.2). The application is then assigned to the device that minimizes the imbalance.

Fig. 2 shows the architectural workflow of Smart-Balance, which operates in two phases:

1. Prediction phase: the scheduler estimates the potential imbalance of assigning an application to each device through the following steps:
   - Users compile their applications with a provided script that extracts key application features during compilation. To preserve privacy, users submit the compiled code together with the extracted features, rather than the source code.
   - The scheduler uses the *Execution Time Predictor*, a pre-trained DL model, to estimate the application's execution time on all available devices based on the extracted features.
   - Using the predicted execution times and device loads, the scheduler computes the potential imbalance for each device and selects the option with the lowest imbalance value.
2. Scheduling phase:
   - The application is assigned to the selected device for execution and the load of the device is updated.

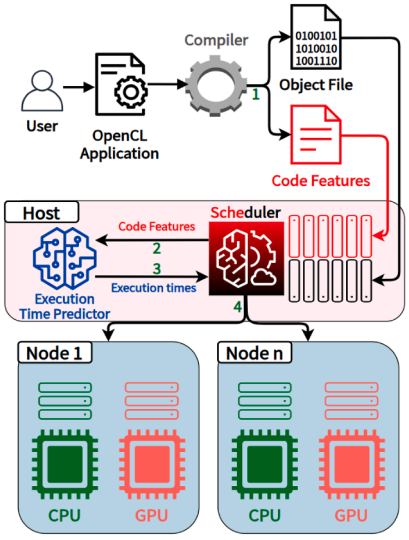


Fig. 2. Smart-balance.

Table 1
Notations.

Symbols	Description
A	Submitted OpenCL applications
F	Code features of the submitted applications
n	Number of the system's devices
$nbrApp$	Total number of applications submitted
$totalTime$	Total time required for executing all submitted applications
$schedQueue$	Scheduler's queue
$Time$	List of predicted execution time of an application on all the system's devices
$Time_{ij}$	Predicted execution time of application i on device j
F_i	List of OpenCL kernel code features of an application 'i'
$Load_j$	Load of node 'j'
Q_j	The number of applications in the queue of the node 'j'
$Balance$	The system's average load
$Imbalance$	The system's imbalance
$Time_{active_j}$	Active time of device 'j'
$Time_{idle_j}$	Idle time of device 'j'
E_{active_j}	The energy consumed by node 'j' per second when executing tasks
E_{idle_j}	The energy consumed by node 'j' per second when idle

3.2. Mathematical model

Table 1 provides the symbols used in describing the mathematical model of the solution.

To calculate the load of a device 'j', we sum the predicted execution times of all applications currently queued for execution on that device. This includes all tasks assigned to device 'j' that has yet to be completed. The formal definition is:

$$Load_j = \sum_{i=0}^{Q_j - 1} Time_{ij} \tag{1}$$

To quantify the imbalance in workload distribution, we propose an *imbalance* metric, defined as follows:

In a balanced system, the load of all devices are equal:

$$Load_1 = Load_2 = \ldots = Load_n \tag{2}$$

By summing the loads of all devices, we obtain:

$$\sum_{j=1}^{n} Load_j = n \times Load_i, \forall i \in [1, n] \tag{3}$$

Consequently, for a system to be balanced, the load on each device must be equal to the average value $Balance$:

$$Balance = \frac{1}{n} \sum_{j=1}^{n} Load_j \tag{4}$$

We define the *"imbalance"* metric as the cumulative absolute deviation of each device's load from the *"balance"* value:

$$Imbalance = \sum_{j=1}^{n} \mid Load_j - Balance \mid \tag{5}$$

After computing the imbalance for all potential scheduling options and selecting the device with the minimum imbalance, the load of the chosen device is updated as follows:

$$Load_{minImabal} = Load_{minImabal} + Time_{minImabal} \tag{6}$$

3.2.1. Optimizing imbalance

Calculating the system imbalance for each candidate device requires iterating over all devices to recompute deviations after each hypothetical assignment. This results in a time complexity of $O(n)$ per device and thus $O(n^2)$ per application, which becomes computationally expensive in large-scale systems.

To reduce this cost, we exploit the fact that, after assigning an application to a device k, only the load of that device and the system balance are modified. This enables us to reuse precomputed sums instead of recalculating the imbalance.

At the beginning of each scheduling decision, we first compute the current balance (*Eq. 4*), and the deviation of each device from the Balance:

$$Dev_j = Load_j - Balance \tag{7}$$

We then partition these deviations by sign and calculate their sums and counts:

Sum of deviations:

$$sumPos = \sum_{D_j > 0} D_j, \qquad sumNeg = \sum_{D_j < 0} D_j \tag{8}$$

Count of positive, negative and zero deviations:

$$c_p = |D_j > 0|, \qquad c_n = |D_j < 0|, \qquad c_z = |D_j = 0| \tag{9}$$

All five values are calculated in the same single loop with a cost $O(n)$.

When application i, with predicted execution time $Time_{i,k}$ is hypothetically assigned to device k, the new load becomes:

$$Load'_k = Load_k + Time_{i,k} \tag{10}$$

Which shifts the balance to:

$$Balance' = Balance + \frac{Time_{i,k}}{n}, \qquad shift = \frac{Time_{i,k}}{n} \tag{11}$$

The deviations of devices k and $j \neq k$ are updated as:

$$Dev'_k = Dev_k + Time_{i,k} - shift \quad Dev'_j = Dev_j - shift \tag{12}$$

The post-assignment imbalance for candidate k is then:

$$Imbalance' = \mid Dev'_k \mid + \sum_{j \neq k} \mid Dev_j - shift \mid \tag{13}$$

Assuming that no devices deviations among $j \neq k$ becomes negative when subtracting the shift:

Each zero Dev_j deviation becomes $shift$

Each positive Dev_j deviation becomes $Dev_j - shift$

Each negative Dev_j deviation becomes $-Dev_j + shift$

$$Imbalance'_{(1)} = \mid Dev_k + Time_{i,k} - shift \mid + shift \cdot c_z^{(-k)} + (sumPos^{(-k)} - shift \cdot c_p^{(-k)}) + (-sumNeg^{(-k)} + shift \cdot c_n^{(-k)}) \tag{14}$$

Where the superscript $(-k)$ indicates that the device k is removed from the corresponding sums and counts.

However, certain devices with $0 < Dev_j < shift$ will cross from positive to non-positive deviation values. For each such device, the deviation term is incorrectly computed as $Dev_j - shift$ instead of the correct value $shift - Dev_j$. The resulting error for each crossing device is:

$$-(Dev_j - shift) + (shift - Dev_j) = 2 \cdot (shift - Dev_j) \tag{15}$$

Summing across all crossing devices, with c_{cross} as their count and $sumCross$ as the sum of their deviations, produces the correction:

$$2 \cdot (c_{cross} \cdot shift - sumCross) \tag{16}$$

Hence, the exact imbalance becomes:

$$Imbalance' = Imbalance'_{(1)} + 2 \cdot (c_{cross} \cdot shift - sumCross) \tag{17}$$

Naively computing c_{cross} and $sumCross$ requires iterating over all positive deviations for each candidate device, which reintroduces the original $O(n^2)$ complexity. To avoid this, we maintain a balanced search tree containing all positive deviations $Dev_j > 0$ along with the count and the sum of its subtree for each node. Constructing the tree requires $O(n \cdot log_n)$, since inserting each of the n devices while updating the corresponding subtree statistics takes $O(log_n)$.

Once the tree is built, querying the number and sum of devices with deviations less than or equal to a given $shift$ can be performed in $O(log_n)$ by traversing the tree from root to leaf. This optimization reduces the imbalance evaluation to $O(n)$ in the common case without crossover deviations, and to $O(n \cdot log_n)$ in the worst case.

To further optimize the algorithm, instead of rebuilding the tree at every scheduling step in $O(n \cdot log_n)$, we exploit the fact that after assigning an application, only the deviation of the chosen device changes explicitly. This update can be applied to the tree in $O(log_n)$. The deviations of all other devices shift uniformly by the same offset $-shift$. Therefore, rather than updating each value individually, we maintain a global shift variable incremented at each scheduling decision, and then the deviation of a device is given by $Dev_j - globalShift$.

3.2.2. Performance metrics

To evaluate the performance of the system, we consider several key metrics:

1. Total Imbalance: This metric calculates the cumulative imbalance throughout the experiment, serving as an indicator of the system's global load balance.

$$totalImbalance = \int_{t_0}^{t_n} imbalance \tag{18}$$

2. Throughput: This metric quantifies the number of applications executed per unit of time [24]. It directly reflects the productivity of the system: a higher throughput indicates that more workloads are completed within the same time frame, meaning that more users are served.

$$throughput = \frac{nbrApp}{totalTime} \tag{19}$$

3. Total Resource Utilization : This metric represents the average utilization percentage of all devices [32].

$$Utilization = \frac{1}{n} \sum_{i=1}^{n} \frac{Time_{active_i}}{totalTime} \tag{20}$$

4. The energy consumption of a device is calculated using its active and idle energy consumption and is measured in Watts.

The active energy of a device 'j' is the total energy consumed by the device when executing computational tasks:

$$Energy_{active_j} = Time_{active_j} \times E_{active_j} \tag{21}$$

The idle energy is the total energy consumed by the device while in an idle state:

$$Energy_{idle_j} = Time_{idle_j} \times E_{idle_j} \tag{22}$$

The total device's energy consumption is:

$$Energy_j = Energy_{active_j} + Energy_{idle_j} \tag{23}$$

Consequently, the total energy consumed by all system's devices is:

$$Energy = \sum_{j=1}^{n} Energy_j \tag{24}$$

3.3. Scheduling algorithm

Algorithm 1: Smart-balance.

Data: Set of applications A and their features F
Result: Device assignment for each application

1 $schedQueue \leftarrow \{A, F\}$;
2 $Sort(schedQueue)$;
3 **while** $schedQueue \neq \emptyset$ **do**
4 $(F_0, app) \leftarrow schedQueue[0]$;
5 $Time \leftarrow ExecutionTimePredictor(F_0)$;
6 % *Step 1: Precompute System State* % ;
7 $initialState(Balance, Dev, sumPos, sumNeg, c_p, c_n, c_z)$;
8 % *Step 2: Building Positive Deviations Tree* %;
9 $Cross \leftarrow \{Dev > 0\}$
10 % *Step 3: Evaluate Each Candidate Device* %;
11 $bestImbalance \leftarrow \infty$;
12 **for** *each device* k **do**
13 $shift \leftarrow \frac{Time_{0,k}}{n}$;
14 $sumCross, c_{cross} \leftarrow query\ (Cross, shift)$;
15 **if** $c_{cross} = 0$ **then**
16 $predictedImbalance \leftarrow$ *Equation (14)*;
18 **end**
19 **else**
20 $predictedImbalance \leftarrow$ *Equation (17)*;
22 **end**
23 **if** $predictedImbalance < bestImbalance$ **then**
24 $bestImbalance \leftarrow predictedImbalance$;
25 $bestDevice \leftarrow k$;
26 **end**
27 **end**
28 % *Step 4: Assign Application To The Best Device* %;
29 $Assign(app, bestDevice, Time[bestDevice])$;
30 % *Option A: rebuild next iteration in* $O(n \cdot log_n)$;
31 % *Option B: incremental update using global shift in* $O(log_n)$;
32 **end**

The Smart-Balance scheduler (Algorithm 1) assigns applications to devices with the objective of minimizing system imbalance. The scheduling process begins by inserting applications and their associated features into a queue and sorting them according to submission time (Lines 1–2). For each application retrieved from the queue, its execution time on all devices is predicted using the *ExecutionTimePredictor*, a pretrained deep learning model designed for this purpose (Line 5).

For each scheduling decision, the scheduler first precomputes the current system state, which includes the balance, the deviation of each device from this balance, and the aggregated sums of positive, negative, and zero deviations (Eqs. 7-9). These values form the foundation for efficient imbalance evaluation (Line 7). The scheduler then stores the set of positive deviations in a balanced search tree (Line 9).

For each candidate device, the scheduler estimates the imbalance that would result if the application were assigned to it using the precomputed sums (see Section 3.2). Specifically, the scheduler queries the tree *Cross* to determine the number and sum of deviations that cross zero after applying the shift (Line 14). If no crossover occurs (Lines 15–16), the imbalance is computed directly using Eq. 14. Otherwise, corrections are applied according to Eq. 17 (Line 19).

Finally, the device with the lowest predicted imbalance is selected (Lines 21–24), the application is assigned to it (Line 27), and the system state is updated accordingly (Line 28). As detailed in the mathematical model in Section 3.2, this update can be further optimized through the use of a global shift (Line 29). This process is repeated until all applications in the scheduling queue have been scheduled.

Table 2
OpenCL's code features.

Features	Extraction	Features	Extraction
Input Data Size	RegExp	Subtractions	LLVM
Returns	LLVM	Function Calls	LLVM
Controls	LLVM	Functions	LLVM
Memory Loads	LLVM	Blocks of Instructions	LLVM
Memory Stores	LLVM	Integer Operations	RegExp
Multiplications	LLVM	Float Operations	RegExp
Divisions	LLVM	Total Instructions	LLVM
Conditions	LLVM	Loop Operations	LLVM
Additions	LLVM	Loops	LLVM

4. Execution time prediction

Building execution time prediction models involves four key phases: dataset preparation, model training, and model testing. The dataset preparation phase involves creating the dataset and then preprocessing it to ensure that it is optimized for the training models. Each computing device in the cluster has its own prediction model tailored to its hardware characteristics, which was trained on a distinct dataset.

4.1. Dataset creation

The dataset used in this study was constructed using the PolyBench benchmark suite [33], a collection of OpenCL parallel applications spanning multiple computational domains, including linear algebra, linear algebra solvers, data mining, and stencil computations. These applications exhibit diverse computational and memory access patterns, which result in varying device suitability under different configurations.

To ensure diversity within the dataset, we executed each each application across a range of input data sizes and recorded its corresponding execution times. This produced a dataset consisting of application's code features paired with its measured execution time, serving as the target variable for training.

4.1.1. Features extraction

To represent each application, we extract a set of 18 OpenCL kernel code features that capture various characteristics impacting the application's execution time on the devices. These features are listed in Table 2.

This process was carried out using a feature extractor script that we developed, employing both Low-Level Virtual Machine (LLVM) pass [34] and regular expressions. LLVM provides a comprehensible intermediate representation (IR) of programs, from which we derive 15 features using a custom LLVM pass. The remaining 3 features are extracted through regular expression parsing directly from the kernel source code. The entire feature extraction workflow is illustrated in Fig. 3.

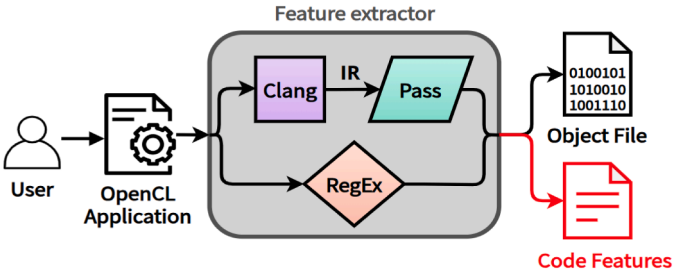


Fig. 3. Code features extractor.

4.1.2. OpenCL applications execution

Each OpenCL application was executed 2000 times per device with varying input data sizes. In total, this produced 40,000 samples for each device, forming the basis for training execution-time prediction models.

4.2. Dataset preprocessing

Data preprocessing transforms raw data into a format suitable for machine learning. This involves data cleaning, formatting, and feature selection. In our study, we used PyCaret [35] library to preprocess our datasets and build ML prediction models.

4.2.1. Features selection

Features selection is the process of identifying and extracting the most relevant features from a dataset that significantly influence the model's predictive performance. This procedure allows reducing the dimensionality of the dataset by excluding redundant and less significant features [36], resulting in several advantages, such as faster training times, improved interpretability, and enhanced prediction accuracy. Furthermore, it reduces the risk of overfitting by reducing the model's exposure to noise and irrelevant data patterns.

In summary, feature selection can improve both prediction accuracy and model robustness. In our study, we adopt three complementary techniques for feature selection to ensure the extraction of the most relevant features:

- Ranking features through Decision Tree Analysis based on their influence on model performance.
- Recursive Feature Selection to identify the optimal number of features that achieve the highest performance of the model.
- Correlation Analysis to validate the selection of features made in the previous phases. This ensures that the chosen features are not redundant.

Decision Tree Features Importance Ranking. Decision tree-based feature importance ranking quantifies the contribution of each feature to the model's predictive capability. Features are assigned a rank based on their importance scores, with the most important features having the highest ranks [36]. The resultant feature ranking is shown in Fig. 4.

Recursive Features Selection. Recursive feature selection (RFS) is an iterative training process that removes one feature from the dataset at each iteration based on its importance score. At each iteration, the model is retrained using the remaining features and its performance is evaluated using the coefficient of determination (R^2). This process continues until no features remain [37], enabling the identification of the most important features having the most significant impact on the model's performance. The results of this process are shown in Fig. 5.

Correlation analysis. Correlation analysis is a statistical method that quantifies the relationship between features within a dataset. It is used to examine the dependencies between these features by identifying those that exhibit strong correlations [24]. Highly correlated features provide

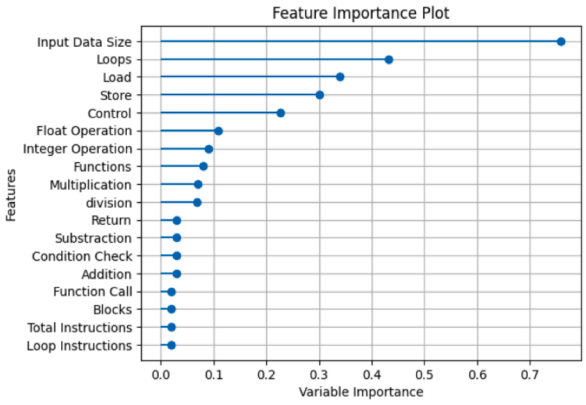


Fig. 4. Features ranking.

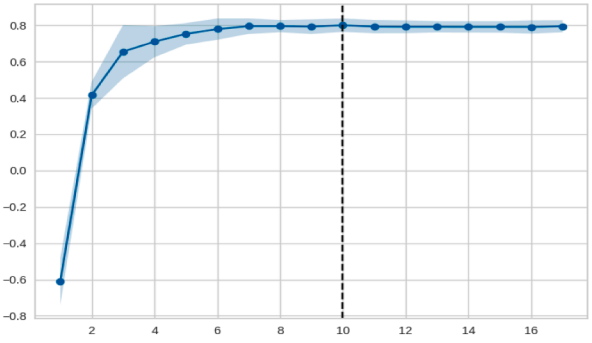


Fig. 5. Recursive features selection.

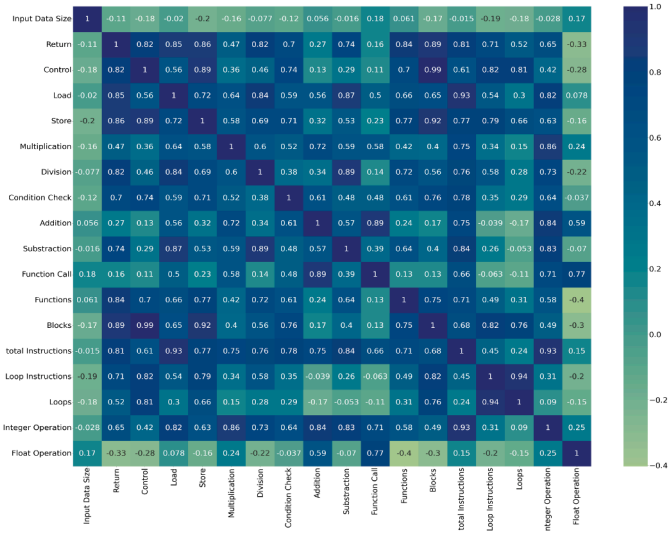


Fig. 6. Correlation matrix.

similar information and cause redundancy, leading to reduced accuracy, increased training time and overfitting. Overfitting occurs when a model is tuned to the training data and cannot generalize to new data. Eliminating correlated features prevents the model from learning the specific patterns of the training data and creates a model that is more accurate and generalizable.

Fig. 6 presents the correlation matrix corresponding to the features listed in Table 2. Notably, it identifies pairs of features exhibiting strong correlations, with correlation values surpassing 0.9. These features are:

- ("Control", "Blocks ")
- ("Load", "Total instructions ")

Table 3
Selected OpenCL features.

No	Features	No	Features
1	Input Data Size	6	Float Operations
2	Number of loops	7	Integer Operations
3	Memory Loads	8	Functions
4	Memory Stores	9	Multiplications
5	Control Instructions	10	Divisions

Table 4
ML time prediction model's testing.

Device	Model	MSE	R2
CPU1	Extra Trees Regressor	1.512 e-3	0.9498
GPU1	Gradient Boosting Regressor	2.916 e-3	0.9529
CPU2	Extra Trees Regressor	7.744 e-3	0.9612
GPU2	Gradient Boosting Regressor	4.489 e-3	0.9427
CPU3	Extra Trees Regressor	2.560 e-3	0.9339
GPU3	Gradient Boosting Regressor	5.320 e-3	0.9273

- ("Store ", "Blocks ")
- ("Total instructions ", "Integer operations ")
- ("Loop instructions ", "Loops ")

4.2.2. Selected features

The results of the recursive feature selection process indicate that the top 10 ranked features produce the highest model accuracy. These features, identified based on their importance scores, represent the optimal subset for prediction. Table 3 presents the list of these top 10 features, as initially ranked through decision tree analysis. Importantly, correlation analysis confirms that these features do not exhibit high intercorrelations, thereby validating their selection and ensuring minimal redundancy within the dataset.

4.3. Models training

In this section, we develop regression models to predict application execution times using both ML and DL techniques. Regression, a fundamental AI method for predicting continuous values, is employed to build these models. The models can be trained on any external machine and later deployed within the system alongside the scheduler. Subsequently, we evaluate and compare the models' performance.

4.3.1. Machine learning prediction models

Each dataset is divided into three distinct subsets: the train set, representing 80% of the dataset, the validation set, representing 10%, and the test set which constitutes of the remaining 10%.

The machine learning models were developed using the PyCaret library. PyCaret automatically trains multiple regression algorithms and compares their performance using cross-validation. The top three performing models, as determined during this phase, were retrained using the combined training and validation sets. Their hyperparameters were optimized using PyCaret's automated tuning functionality. The final results of the best models on the test set are presented in the Table 4. This tables report the mean squared error (MSE) and coefficient of determination (R^2) values for the test sets.

The MSE metric evaluates the prediction accuracy by computing the average of the squared differences between the predicted and actual values. On the other hand, the R-squared metric indicates how effectively models represent the data. A higher R-squared value indicates a better fit, while minimal MSE value indicates increased accuracy [38]. Therefore, for each device, the best prediction model has the highest value of R-squared and the lowest value of MSE.

4.3.2. Deep learning prediction models

To implement the neural network prediction models, we employed Keras library [39]. Keras provides a high-level interface for building and training deep learning (DL) models. It is built on top of TensorFlow [40], a widely-used framework that offers robust tools for building and training DL models.

In our experiment, we configured each deep learning model with the following training parameters: 500 epochs, Adam optimizer, a learning rate of 0.001, and a batch size of 64.

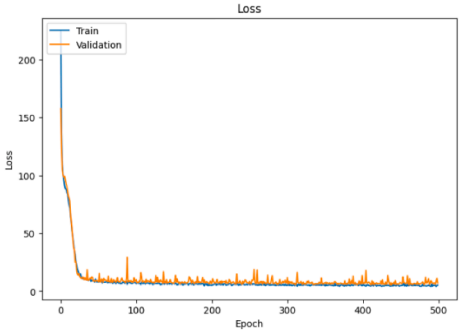


Fig. 7. CPU1 loss curve.

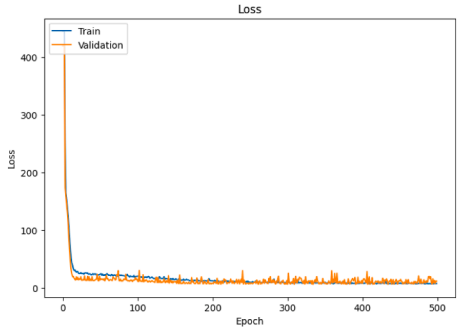


Fig. 8. GPU1 loss curve.

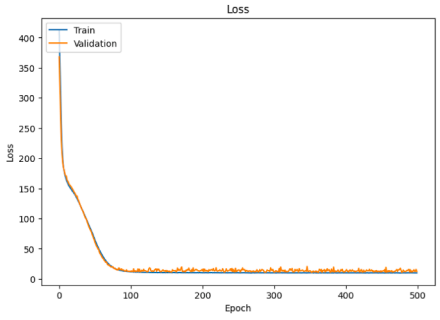


Fig. 9. CPU2 loss curve.

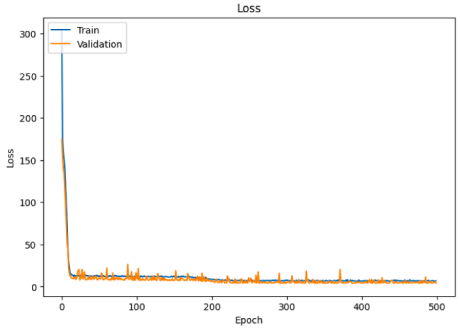


Fig. 10. GPU2 loss curve.

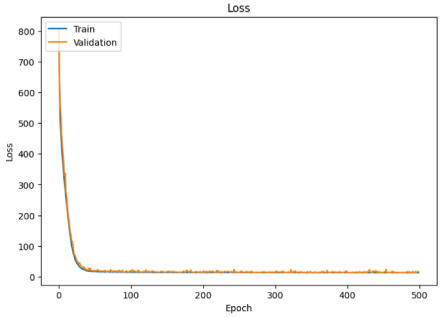


Fig. 11. CPU3 loss curve.

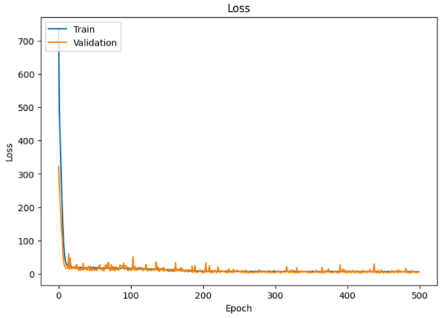


Fig. 12. GPU3 loss curve.

Table 5
DL models architecture.

Layer	CPU1	GPU1	CPU2	GPU2	CPU3	GPU3
1	10	10	10	10	10	10
2	80	60	80	60	80	60
3	75	50	75	50	70	55
4	1	1	1	1	1	1

Table 6
DL time prediction model's testing.

	MSE	R2
CPU1	1.3456 e-3	0.9633
GPU1	2.500 e-3	0.9684
CPU2	2.916 e-3	0.9787
GPU2	3.844 e-3	0.9523
CPU3	2.528 e-3	0.9497
GPU3	5.103 e-3	0.9441

The architecture of the deep multilayer perceptron (DMLP) models is presented in Table 5. Each model uses the Rectified Linear Unit (ReLU) function as the activation function in its hidden layers. The hyperparameters of these models were manually fine-tuned to achieve optimal performance.

To evaluate the performance of the deep learning models during training, we used the Mean Squared Error (MSE) as the loss function. Figs. 7–12 illustrate the training and validation loss curves for the deep learning prediction models. The loss curves for the training set (blue) and the validation set (orange) show a consistent decrease over time and converge towards zero, suggesting that the models effectively learn from the training data. The close alignment between the two curves indicates that the models generalize well to unseen data, without signs of overfitting or underfitting.

To assess the final performance of the DL models, we evaluated them on the test set, representing new and unseen data for the models. The evaluation's results are presented in Table 6.

In summary, the deep learning (DL) prediction models outperformed the traditional machine learning (ML) models, achieving lower Mean Squared Error (MSE) and higher R-squared values. These results indicate that the DL models provide a more accurate representation of the data and deliver more precise predictions compared to the ML models.

5. Performance evaluation

We deployed our scheduler on a multi-node heterogeneous cluster consisting of multiple CPUs and GPUs. We conducted a series of experiments with various workload distributions to assess the scheduler's performance and ability to adapt and function efficiently in real-world scenarios.

5.1. Experimental setup

5.1.1. System

The system consists of a master host and three worker servers, each equipped with a CPU and a GPU. The master host is responsible for scheduling tasks across the worker nodes. The detailed specifications of the cluster are provided in Table 7.

5.1.2. Workload

We evaluated 20 OpenCL applications with five input sizes, resulting in 100 distinct jobs. Reported results are averaged over 51 independent runs to ensure reliability and average stability.

5.1.3. Scenarios

To emulate realistic computing scenarios, we designed three workload types based on application computational demands, primarily determined by input data sizes:

- Moderate workload: applications with relatively small input sizes (1,000,000 – 10,000,000 float elements), generally more suitable for CPUs.
- Heavy workload: applications with larger input sizes (10,000,000 - 100,000,000 float elements), typically better suited for GPUs.
- Mixed workload: a combination of applications from both moderate and heavy workloads.

5.1.4. Baselines

To assess the performance of our proposed scheduling technique, we compared it against several baseline scheduling strategies, including some that were adapted from other fields:

1. Round Robin (RR)
   Round robin allocates applications to the devices in rotational order. It was implemented in the works [6],[28],[30].
2. Weighted Round Robin (WRR)
   WRR assigns weights to devices based on their processing capabilities. Devices with higher weights execute more tasks. WRR was implemented in [6],[28], and [29].

- Calculating devices' weights: Each computing device is assigned a weight proportionally to its computing capabilities. The sum of weight of all devices is equal to one [28]. In our system, we calculate the weight of each device based on its number of cores and frequency. The weight of a device is calculated following Eq. 16.

$$Weight = \frac{\frac{Number\,of\,Cores}{Total\,Number\,of\,Cores} + \frac{Total\,Number\,of\,Cores}{Total\,Frequency}}{2} \tag{25}$$

Table 8 presents the weight calculation for the system devices. The values were rounded to two decimal places.

According to the weight distribution, in WRR:

- CPU1, CPU2, CPU3 each receive one application in every round.
- GPU3 takes 2 applications in every round.

Table 7
Heterogeneous cluster.

Node	Operating System	Ram	CPU	GPU
Master	Ubuntu Desktop 22.04.1 LTS	8 GB	Intel® Core™ i5-6300U	Intel HD Graphics 520 graphics card
Worker 1	Ubuntu Server 18.04.1 LTS	8 GB	Intel® Xeon® E3-1225 v6	NVIDIA GM107GL Quadro P620
Worker 2	Ubuntu Server 18.04.1 LTS	8 GB	Intel® Xeon® E3-1225 v6	NVIDIA GM107GL Quadro P400
Worker 3	Ubuntu Server 18.04.1 LTS	8 GB	Intel® Core™ i7-6700U	NVIDIA GM107GL Quadro K400

Table 8
Weight calculation.

Device	Model	Cores	Frequency	Weight	$\frac{Weight}{WeightMin}$
CPU1	E3-1225 [41]	4	3.70	0.13	1
GPU1	P400 [42]	256	1.252	0.18	1.5
CPU2	E3-1225 [41]	4	3.7	0.13	1
GPU2	P400 [42]	256	1.252	0.18	1.5
CPU3	i7-6700 [43]	8	4	0.14	1
GPU3	K620 [44]	384	1.124	0.25	2

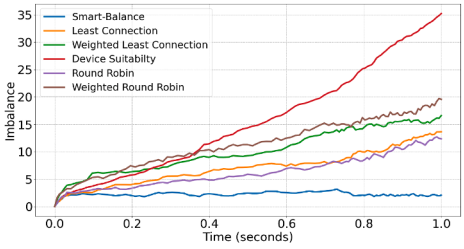


Fig. 13. Imbalance curve for moderate workload.

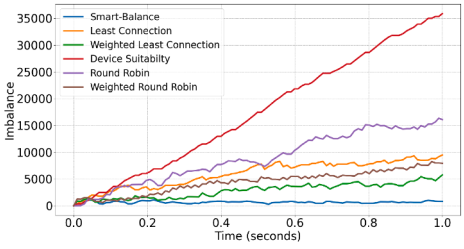


Fig. 14. Imbalance curve for heavy workload.

- GPU1 and GPU2 each take three applications in every two rounds, alternating between receiving one and then two applications across successive rounds.

3. Device Suitability (DS)
   We employed our DL models to predict the execution time of each task across all available devices and subsequently assigned each task to the fastest device. This approach was adopted in [20],[21],[22],[23],[25], and [45].
4. Least Connections (LC)
   In our version of LC, it assigns tasks to the device with the least number of queued applications instead of assigning them to the server with the least active connections. This method was presented in [6] and [30].
5. Weighted Least Connection (WLC)
   This strategy considers both the number of applications in a device's queue and its weight. Applications are assigned to the device with the minimum ratio of Connections to Weight. This method was implemented in [31].

5.1.5. Metrics

We compared the performance of our proposed approach against the baseline methods using the following metrics:

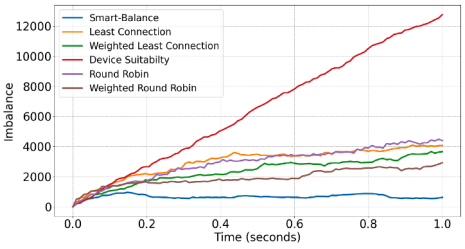


Fig. 15. Imbalance curve for mixed workload.

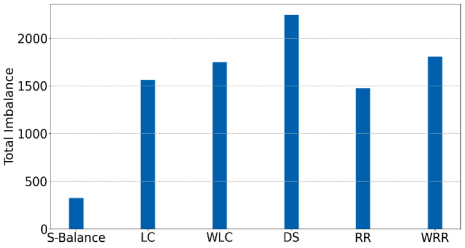


Fig. 16. Total imbalance for moderate workload.

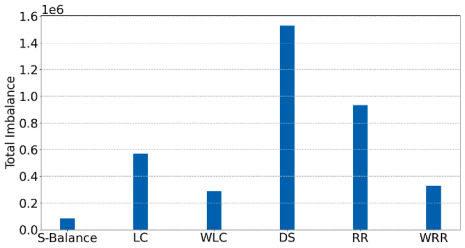


Fig. 17. Total imbalance for heavy workload.

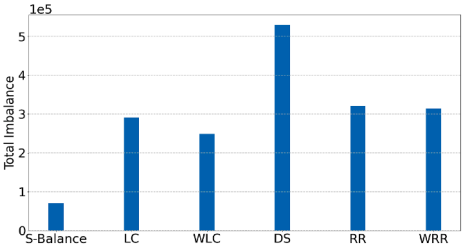


Fig. 18. Total imbalance for mixed workload.

1. "Imbalance": This metric represents the imbalance of the system at a specific time during the experiment.
2. "Total imbalance": Represents the cumulative load imbalance over the entire duration of the experiment. A lower total imbalance value indicates a more efficient load balance throughout the whole experiment.
3. "Finishing time": Represents the total time required to execute the entire workload in the system. It indicates how quickly the system can process all the tasks assigned to it.

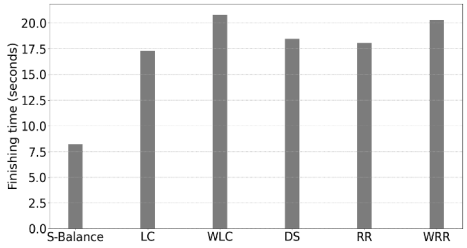


Fig. 19. Finishing time for moderate workload.

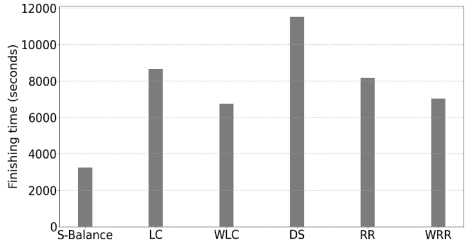


Fig. 20. Finishing time for heavy workload.

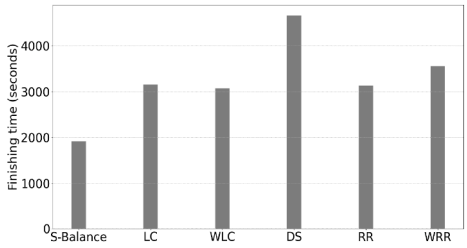


Fig. 21. Finishing time for mixed workload.

4. "Device active time": Refers to the time taken by each device to finish all applications assigned to it. This metric is useful for evaluating how workloads are distributed across devices.
5. "Device idle time": Represents the time during which each device remains inactive without performing any computational tasks. This metric serves as an indicator of resource underutilization and potential energy wastage.
6. "Energy consumption": This metric measures the total energy consumed by all devices during the experiment.
7. "Throughput": this metric represents the number of applications executed per second. Higher throughput indicates a more productive system.
8. "System Utilization": Represents the percentage of time during which all devices are actively performing computations, providing an overall measure of system efficiency.

5.2. Experimental results

When reporting the results of Smart-Balance (S-Balance), Least Connection (LC), Weighted Least Connection (WLC), Device Suitability (DS), Round Robin (RR), and Weighted Round Robin (WRR) approaches, the overhead associated with execution time prediction was considered.

5.2.1. Imbalance

Figs. 13–15 present the system's load imbalance curves for the different approaches across the three workload scenarios. The times have been normalized to allow a clear comparison among the five schedulers.

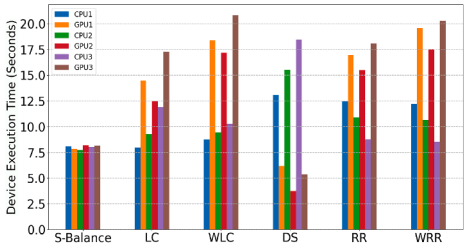


Fig. 22. Device active time for moderate workload.

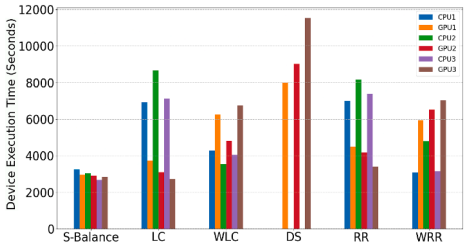


Fig. 23. Device active time for heavy workload.

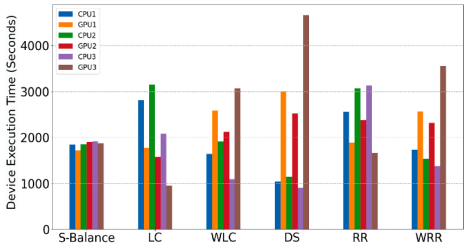


Fig. 24. Device active time for mixed workload.

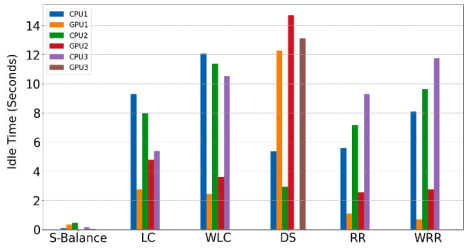


Fig. 25. Device idle time for moderate workload.

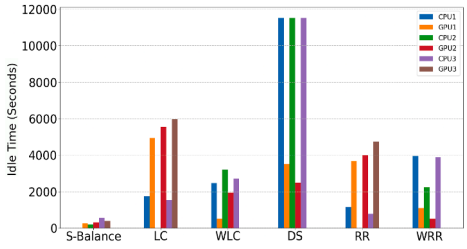


Fig. 26. Device idle time for heavy workload.

The Smart-Balance approach consistently converges to zero load imbalance as the number of applications increases. In contrast, the other approaches exhibit a gradual increase in load imbalance over time. These results highlight the effectiveness of Smart-Balance in maintaining a balanced workload.

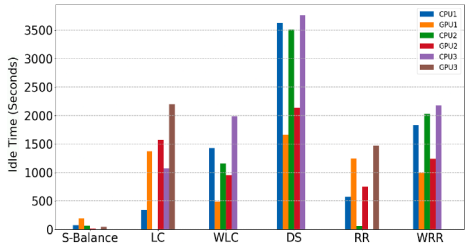


Fig. 27. Device idle time for mixed workload.

5.2.2. Total imbalance

Figs. 16–18 illustrate the total accumulated load imbalance throughout the experiments. The Smart-Balance approach outperforms all other methods, achieving the lowest total system imbalance in every scenario.

- Smart-Balance reduces the total imbalance by 79%, 81%, 86%, 78%, 82% less than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for moderate workloads.
- Smart-Balance reduces the total imbalance by 76%, 72%, 86%, 78%, 70% less than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for heavy workloads.
- Smart-Balance reduces the total imbalance by 76%, 71%, 86%, 78%, 70% less than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for mixed workloads.

5.2.3. Finishing time

Figs. 19–21 illustrate the total execution times required to finish the execution of the submitted workloads. In all three scenarios, the Smart-Balance approach achieves the shortest execution time for all submitted workloads.

- Smart-Balance is 2.1, 2.5, 2.25, 2.2, 2.47 times faster than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for moderate workloads.
- Smart-Balance is 2.6, 2, 3.5, 2.5, 2.1 times faster than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for heavy workloads.
- Smart-Balance is 1.6, 1.6, 2.4, 1.6, 1.9 times faster than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for mixed workloads.

5.2.4. Device active time

Figs. 22–24 depict the active time of each computing device. The Smart-Balance approach achieved an equitable distribution of loads across the devices, as the active times of the devices are close to equal. In contrast, the other approaches exhibited overloads in all three scenarios.

5.2.5. Device idle time

Figs. 25–27 display the idle time of devices throughout the experiments. Longer idle times indicate wasted energy and imbalanced workload distribution.

Smart-Balance consistently achieves minimal idle times across all devices, demonstrating its ability to fully exploit the available computational resources. Conversely, other methods exhibit significant idle times, indicating suboptimal resource utilization and energy inefficiency.

5.2.6. Energy consumption

Energy consumption was measured using *"perf"*[46] for Intel devices and *"nvidia-smi"*[47] for NVIDIA devices. Given the dynamic factors

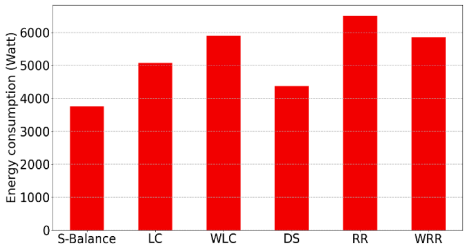


Fig. 28. Energy for moderate workload.

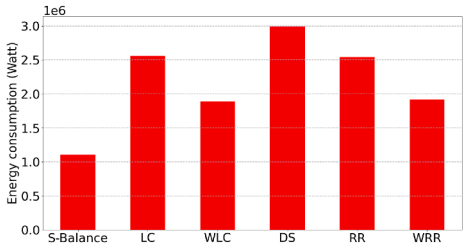


Fig. 29. Energy for heavy workload.

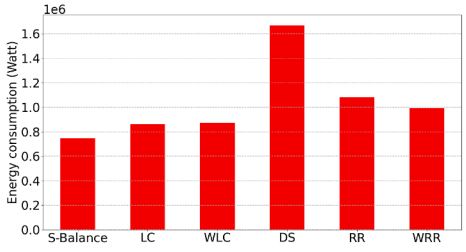


Fig. 30. Energy for mixed workload.

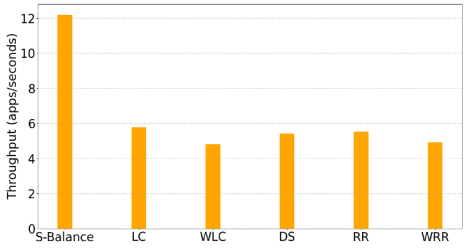


Fig. 31. System throughput for moderate workload.

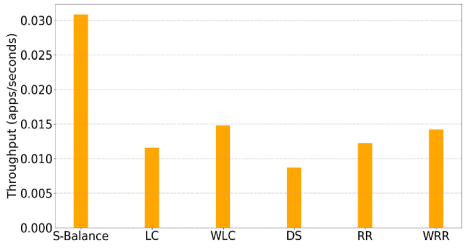


Fig. 32. System throughput for heavy workload.

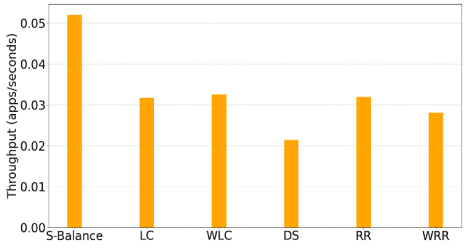


Fig. 33. System throughput for mixed workload.

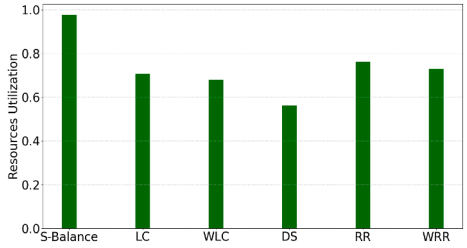


Fig. 34. System utilization for moderate workload.

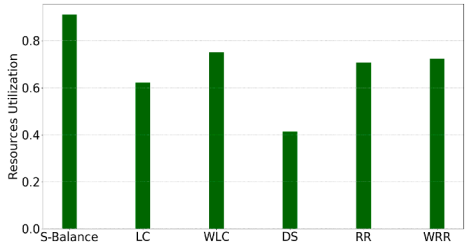


Fig. 35. System utilization for heavy workload.

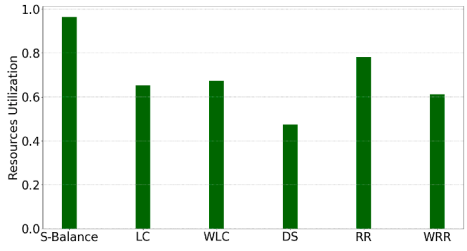


Fig. 36. System utilization for mixed workload.

that influence power consumption, such as connected peripherals, background processes, and operating system configurations, the reported values represent the mean of multiple measurements conducted under identical conditions.

Figs. 28–30 present energy consumption:

- Smart-Balance reduces the energy consumption by 38%, 45%, 44%, 43%,47% less than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for moderate workloads.
- Smart-Balance reduces the energy consumption by 56%, 42%, 63%, 56%, 42% less than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for heavy workloads.
- Smart-Balance reduces the energy consumption by 42%, 33%, 53%, 46%, 32% less than Least Connections, Weighted Least Connections, Device Suitability, Round Robin and Weighted Round Robin respectively for mixed workloads.

5.2.7. Throughput

Figs. 31–33 depict the throughput of the system. Across all scenarios, Smart-Balance consistently achieves the highest throughput, enabling the system to process more applications concurrently and efficiently.

5.2.8. System utilization

Figs. 34–36 illustrate resources utilization. Smart-Balance achieves the highest overall system utilization in every scenario, indicating its effectiveness in keeping computing resources engaged in computing and minimizing idle periods.

5.3. Results discussion

According to the experimental results reported in Figs. 13–36, Smart-Balance consistently outperformed alternative scheduling schemes in terms of load balance, execution time, throughput, resource utilization, and energy efficiency across all tested scenarios. These results demonstrate its robustness and adaptability to dynamic workloads in heterogeneous systems, highlighting its potential applicability in real-world deployments.

In this section, we analyze the behavior and performance of each scheduling approach across the three workload scenarios.

5.3.1. Least connections

The LC algorithm assigns each application to the device with the fewest pending tasks, prioritizing devices that complete tasks faster. However, this algorithm defines a device's load solely based on the number of queued applications, without accounting for task-specific or device-specific characteristics. It assumes that all tasks are equal and all devices operate at the same processing rate. This assumption introduces significant limitations in heterogeneous environments.

For moderate workloads, applications typically run faster on the CPU compared to the GPU, primarily due to the higher cost of data transfer outweighing the speedup gained from GPU acceleration. Consequently, GPUs are overloaded with unsuitable applications since the LC algorithm fails to consider application-to-device affinity (Fig. 22).

In contrast, under heavy workloads, GPUs outperform CPUs significantly. Nevertheless, the LC algorithm continues allocating excessive tasks to CPUs regardless, leading to their overloading while GPUs remain underutilized (Fig. 23).

On the other hand, for workloads with applications having varying computational demands, the LC approach tends to overload CPUs, as heavy applications are frequently assigned to them, resulting in higher CPU utilization (Fig. 24).

The LC approach consistently caused load imbalance across all three scenarios, primarily due to the overloading of unsuitable devices. This highlights a fundamental limitation of the LC approach, which does not differentiate between application characteristics and their compatibility with computing devices.

In heterogeneous systems, devices possess varying computing capabilities. As a result, a device with a large number of applications may still be underutilized if they are suitable to its capabilities. Conversely, a device with fewer applications may be overloaded if applications are incompatible with it.

Therefore, the LC scheduling approach is more likely to produce favorable results in homogeneous systems having identical computing devices executing similar applications, rather than in dynamic, heterogeneous environments.

5.3.2. Weighted least connections

The Weighted Least Connections algorithm extends the Least Connections approach by assigning weights to devices based on their computational power. Applications are then allocated to devices with the lowest connections-to-weight ratio, favoring devices with higher computational power, such as GPUs. However, the WLC algorithm inherits the limitations of the LC approach by maintaining a simplistic load definition. It assumes that devices with higher weights always outperform others, regardless of their suitability with applications.

In the first scenario, GPUs are overloaded with unsuitable applications due to their higher weights and unsuitability for moderate applications. This results in a higher system imbalance compared to the LC approach, as depicted in Fig. 22.

In the second case, the WLC algorithm slightly reduces imbalance by assigning heavier applications to GPUs, which are more suited for such tasks. However, GPUs are still overloaded as they receive a disproportionate number of tasks (Fig. 23).

Similarly, the WLC approach results in GPU overloading for mixed workloads, as the higher weight of GPUs increases the likelihood of assigning them more non-suitable and heavy applications, as shown in Fig. 24.

As a result, the WLC scheduling approach is more effective in homogeneous systems where devices are of the same type but with varying computational power, and workloads consisting of applications with high computing intensity.

5.3.3. Device suitability

The Device Suitability scheduler selects the device that executes the application most quickly, ignoring real-time queue lengths. This approach minimizes the applications' individual execution times but leads to overloading the faster devices.

In scenarios with moderate workloads, CPUs can execute applications more quickly than GPUs. Consequently, the Device Suitability scheduler tends to allocate a higher number of applications to CPUs, overloading them, as shown in Fig. 22. Conversely, in scenarios with a heavy workload, the Device Suitability scheduler results in severe overloading of GPUs while neglecting CPUs as shown in Fig. 23. Whereas in scenarios with a mixed workload, containing both heavy and moderate applications, the Device Suitability approach overloads GPUs with heavy applications, while executing moderate applications on CPUs, as illustrated in Fig. 24. Consequently, the Device Suitability scheduler can be more effective with balanced workloads tailored to device strengths.

5.3.4. Round Robin

The Round Robin scheduler assigns applications to devices in a cyclical manner to the next available device without considering any factors. In this approach, all devices in the system handle an equal number of applications.

In the first scenario (Fig. 22), the Round Robin scheduler tends to overload GPUs, as CPUs are better suited for moderate applications and complete their allocated load more quickly. This imbalance in workload distribution becomes more pronounced when the workload includes intensive applications. CPUs, which are less suited for heavy loads, are often assigned intensive applications, leading to their overloading in the majority of cases, as depicted in Figs. 23 and 24. In summary, Round Robin is suitable for homogeneous systems with identical computing devices and applications.

5.3.5. Weighted Round Robin

The Weighted Round Robin algorithm allocates applications to devices alternatively, with a higher number of applications assigned to devices possessing higher weights in each round.

When the workload is of moderate intensity, the Weighted Round Robin approach overloads GPUs with a larger number of non-suitable applications (Fig. 22). In contrast, in scenarios two and three, which include intensive applications, the Weighted Round Robin scheduler reduces the load imbalance by prioritizing the allocation of these applications to GPUs, as demonstrated in Figs. 23 and 24. Nonetheless, GPUs within the system remain overloaded.

The Weighted Round Robin scheduling approach is better suited for homogeneous systems consisting of similar devices having varying computational power. It is most effective when managing workloads consisting of uniform applications with identical computational intensity.

5.3.6. Smart-balance

The experimental results highlight the limitations of the discussed scheduling approaches when deployed in dynamic and heterogeneous systems. These limitations stem from the simplistic definitions of device load that overlook crucial factors influencing application performance, such as computational intensity, parallelism, device capabilities, and the real-time system state. As a result, the scheduler fails to adapt to changing workloads and make suboptimal scheduling decisions.

The imbalance caused by inadequate scheduling (Figs. 22–24) led to extended execution times. Overloaded devices took longer to complete their excessive workload (Figs. 19–21), leading to increased idle time for underloaded devices (Figs. 25–27). Consequently, the system exhibited low throughput (Figs. 31–33) and resource utilization (Figs. 34–36), resulting in higher energy consumption (Figs. 28–30).

In this work, we address the limitations of the discussed approaches by balancing the workload distribution within the system. We introduce a comprehensive load definition that accurately quantifies the load on each device, along with an imbalance metric that reflects the state of workload distribution in the system. Our scheduler continuously monitors the system's state in real-time and dynamically updates device loads. Upon receiving a new application, the scheduler evaluates the load distribution across all scheduling scenarios and assigns the application to the device that minimizes system imbalance.

Prioritizing devices with lower imbalance leads to a more equitable distribution of workloads, as shown in Figs. 16–18. This selection strategy is reflected in the convergence of the imbalance curves toward zero, highlighting the progressive reduction in system imbalance as more applications are assigned (Figs. 13–15).

Achieving a balanced load distribution resulted in significant performance improvements, including reduced execution time (Figs. 19–21) and idle time (Figs. 25–27) due to synchronous device completion. The optimized device utilization and reduced overloads enhanced throughput, enabling the system to accommodate a larger number of user applications. These combined effects contributed in minimizing energy waste, resulting in more efficient energy consumption (Figs. 28–30), an effect that becomes especially evident as workload grew more computationally intensive, where incorrect scheduling amplifies imbalance and results in significantly higher power consumption.

In our study, we leveraged the inherent characteristics of heterogeneous systems to optimize performance. By identifying the critical factors influencing application execution, we were able to continuously monitor system devices and strategically allocate applications to those that maintain load balance. Notably, our approach demonstrates the ability to adapt to dynamic workloads and heterogeneous devices, consistently delivering high performance across all scenarios.

5.3.7. Overhead

The overhead resulting from feature extraction and execution time prediction is minimal, having a negligible impact on system performance. This outcome is attributed to the seamless integration of feature extraction into the application compilation process. Furthermore, the execution time prediction models, which are trained only once, can be directly applied during their usage period without requiring retraining, ensuring efficient performance throughout the system's operation.

6. Conclusion

Load balancing computing workloads is essential for the stability and performance of computing systems, particularly for real-time heterogeneous systems, where applications are time-sensitive and prior workload information is unavailable.

This work introduced Smart-Balance, a real-time scheduling approach for heterogeneous clusters. It leverages AI-based prediction to monitor system loads and assign applications to the optimal device. This ability is enabled by our novel imbalance metric, which quantifies the disparities in device loads.

When tested against popular schedulers, our approach achieved lower load imbalance, reduced execution times, improved energy efficiency, and higher overall throughput.

Despite its effectiveness, Smart-Balance exhibits certain limitations. It is platform-dependent, requiring prediction models to be retrained for different hardware configurations. Moreover, its performance is tied to the accuracy of its predictions. Finally, scaling the approach to

very large clusters remains a challenge, since existing frameworks (e.g., CUDA, OpenCL) lack built-in support for global scheduling.

To address the limitations of the "Smart-Balance" approach, we propose the following avenues for future work:

- Generalizing prediction models: developing a prediction model for various devices enables "Smart-Balance" to be directly adopted in different hardware configurations.
- Enhancing prediction accuracy: training the model with a broader spectrum of applications ensures the ability to handle diverse computational workloads with higher precision.
- Improving scalability: exploring higher-level abstractions or middleware support for large-scale heterogeneous systems.

CRediT authorship contribution statement

Taha Abdelazziz Rahmani: Writing – original draft, Software, Methodology; **Ghalem Belalem:** Writing – review & editing, Supervision; **Sidi Ahmed Mahmoudi:** Writing – review & editing, Supervision; **Omar Rafik Merad-Boudia:** Writing – review & editing.

Data availability

Data will be made available on request.

Declaration of competing interest

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

References

[1] A. Hayat, Y.N. Khalid, M.S. Rathore, M.N. Nadir, A machine learning-based resource-efficient task scheduler for heterogeneous computer systems, J. Supercomput. 79 (14) (2023) 15700–15728.

[2] T.A. Rahmani, G. Belalem, S.A. Mahmoudi, O.R. Merad-Boudia, Equalizer: energy-efficient machine learning-based heterogeneous cluster load balancer, Concur. Comput. Practice Exp. 36 (23) (2024) e8230.

[3] A. Bestavros, WWW Traffic reduction and load balancing through server-based caching, IEEE Concurrency 5 (1) (1997) 56–67.

[4] L.-H. Hung, C.-H. Wu, C.-H. Tsai, H.-C. Huang, Migration-based load balance of virtual machine servers in cloud computing by load prediction using genetic-based methods, IEEE Access 9 (2021) 49760–49773.

[5] T.A. Rahmani, G. Belalem, S.A. Mahmoudi, O.R. Merad-Boudia, Machine learning-driven energy-efficient load balancing for real-time heterogeneous systems, Cluster Comput. 27 (4) (2024) 4883–4908.

[6] Y.M. Teo, R. Ayani, Comparison of load balancing strategies on cluster-based web servers, Simulation 77 (5–6) (2001) 185–195.

[7] T. Helmy, S. Al-Azani, O. Bin-Obaidellah, A machine learning-based approach to estimate the CPU-burst time for processes in the computational grids, in: 2015 3Rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS), IEEE, 2015, pp. 3–8.

[8] R. Karthikeyan, V. Balamurugan, R. Cyriac, B. Sundaravadivazhagan, COSCO2: AI-Augmented evolutionary algorithm based workload prediction framework for sustainable cloud data centers, Trans. Emerg. Telecommun. Technol. 34 (1) (2023) e4652.

[9] R. Karthikeyan, S.R. Abdul Samad, V. Balamurugan, S. Balasubaramanian, R. Cyriac, Workload prediction in cloud data centers using complex-Valued spatio-Temporal graph convolutional neural network optimized with gazelle optimization algorithm, Trans. Emerg. Telecommun. Technol. 36 (3) (2025) e70078.

[10] J. Zhou, U.K. Lilhore, T. Hai, S. Simaiya, D.N.A. Jawawi, D.M. Alsekait, S. Ahuja, C. Biamba, M. Hamdi, et al., Comparative analysis of metaheuristic load balancing algorithms for efficient load balancing in cloud computing, J. Cloud Comput. 12 (1) (2023) 1–21.

[11] J. Lee, M. Samadi, Y. Park, S. Mahlke, Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems, in: Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques, IEEE, 2013, pp. 245–255.

[12] J.E. Stone, D. Gohara, G. Shi, OpenCL: a parallel programming standard for heterogeneous computing systems, Comput. Sci. Eng. 12 (3) (2010) 66.

[13] B. Pérez, E. Stafford, J.L. Bosque, R. Beivide, Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems, J. Parallel Distrib. Comput. 157 (2021) 30–42.

[14] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, J Parallel Distrib. Comput. 7 (2) (1989) 279–301.

[15] O. Valery, P. Liu, J.-J. Wu, A collaborative CPU–GPU approach for principal component analysis on mobile heterogeneous platforms, J Parallel Distrib. Comput. 120 (2018) 44–61.

[16] J. Planas, R.M. Badia, E. Ayguadé, J. Labarta, SSMART: Smart scheduling of multi-architecture tasks on heterogeneous systems, in: Proceedings of the Second Workshop on Accelerator Programming Using Directives, 2015, pp. 1–11.

[17] B. Pérez, E. Stafford, J.L. Bosque, R. Beivide, Energy efficiency of load balancing for data-parallel applications in heterogeneous systems, J. Supercomput. 73 (1) (2017) 330–342.

[18] S. Memeti, S. Pllana, Optimization of heterogeneous systems with AI planning heuristics and machine learning: a performance and energy aware approach, Computing 103 (12) (2021) 2943–2966.

[19] T. Geng, M. Amaris, S. Zuckerman, A. Goldman, G.R. Gao, J.-L. Gaudiot, A profile-based AI-assisted dynamic scheduling approach for heterogeneous architectures, Int. J. Parallel Program. 50 (1) (2022) 115–151.

[20] I. Harichane, S.A. Makhlouf, G. Belalem, KubeSC-RTP: smart scheduler for kubernetes platform on CPU-GPU heterogeneous systems, Concurr. Comput. Practice and Exp. 34 (21) (2022) e7108.

[21] Y. Wen, Z. Wang, M.F.P. O'boyle, Smart multi-task scheduling for openCL programs on CPU/GPU heterogeneous platforms, in: 2014 21St International Conference on High Performance Computing (HiPC), IEEE, 2014, pp. 1–10.

[22] M. López, E. Stafford, J.L. Bosque, Intelligent energy pairing scheduler (inEPS) for heterogeneous HPC clusters, J. Supercomput. 81 (2) (2025) 1–23.

[23] Z. Bai, D. Wu, P. Dangi, D. Wijerathne, V.P.K. Miriyala, T. Mitra, Data-aware Dynamic Execution of Irregular Workloads on Heterogeneous Systems, arXiv preprint arXiv:2502.06304 (2025).

[24] Y.N. Khalid, M. Aleem, U. Ahmed, M.A. Islam, M.A. Iqbal, Troodon: a machine-learning based load-balancing application scheduler for CPU–GPU system, J. Parallel Distrib. Comput. 132 (2019) 79–94.

[25] U. Ahmed, J.C.-W. Lin, G. Srivastava, M. Aleem, A load balance multi-scheduling model for openCL kernel tasks in an integrated cluster, Soft Comput. 25 (1) (2021) 407–420.

[26] T.A. Rahmani, G. Belalem, S.A. Mahmoudi, RTLB_Sched: real time load balancing scheduler for CPU-GPU heterogeneous systems, in: 2023 International Conference on Smart Computing and Application (ICSCA), IEEE, 2023, pp. 1–6.

[27] T.A. Rahmani, F. Daham, G. Belalem, S.A. Mahmoudi, HBalancer: A machine learning based load balancer in real time CPU-GPU heterogeneous systems, in: 2022 International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), IEEE, 2022, pp. 674–679.

[28] A. Manirabona, S. Boudjit, L.C. Fourati, A priority-weighted round robin scheduling strategy for a WBAN based healthcare monitoring system, in: 2016 13Th IEEE Annual Consumer Communications & Networking Conference (CCNC), IEEE, 2016, pp. 224–229.

[29] W. Wang, G. Casale, Evaluating weighted round robin load balancing for cloud web services, in: 2014 16Th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE, 2014, pp. 393–400.

[30] B. Alankar, G. Sharma, H. Kaur, R. Valverde, V. Chang, Experimental setup for investigating the efficient load balancing algorithms on virtual cloud, Sensors 20 (24) (2020) 7342.

[31] G. Singh, K. Kaur, An improved weighted least connection scheduling algorithm for load balancing in web cluster systems, International Research Journal of Engineering and Technology (IRJET) 5 (3) (2018) 6.

[32] U. Ahmed, M. Aleem, Y. Noman Khalid, M. Arshad Islam, M. Azhar Iqbal, RALB-HC: A resource-aware load balancer for heterogeneous cluster, Concurr. Comput. Pract. Exp. 33 (14) (2021) e5606.

[33] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, J. Cavazos, Auto-tuning a high-level language targeted to GPU codes, in: 2012 Innovative Parallel Computing (InPar), Ieee, 2012, pp. 1–10.

[34] A. Sampson, LLVM for Grad Students, 2015. https://www.cs.cornell.edu/~asampson/blog/llvm.html.

[35] M. Ali, PyCaret: An open source, low-code machine learning library in Python, 2020. PyCaret version 1.0.0, https://www.pycaret.org.

[36] R.-C. Chen, C. Dewi, S.-W. Huang, R.E. Caraka, Selecting critical features for data classification based on machine learning methods, J. Big Data 7 (1) (2020) 52.

[37] X.-w. Chen, J.C. Jeong, Enhanced recursive feature elimination, in: Sixth International Conference on Machine Learning and Applications (ICMLA 2007), IEEE, 2007, pp. 429–435.

[38] D. Chicco, M.J. Warrens, G. Jurman, The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation, PeerJ Comput. Sci. 7 (2021) e623.

[39] MIT, Keras Documentation, https://keras.io/api/.

[40] Google, Tensorflow framework, https://www.tensorflow.org/.

[41] Intel Xeon E3-1225, https://ark.intel.com/content/www/fr/fr/ark/products/52270/intel-xeon-processor-e31225-6m-cache-3-10-ghz.html.

[42] NVIDIA QUADRO P400, https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p400-data-sheet-us-nv-704503-r1.pdf.

[43] Processeur Intel Core™ i7-6700, https://ark.intel.com/content/www/fr/fr/ark/products/88196/intel-core-i76700-processor-8m-cache-up-to-4-00-ghz.html.

[44] Nvidia, NVIDIA QUADRO K620, https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/documents/75509_DS_NV_Quadro_K620_US_NV_HR.pdf.

[45] H.J. Choi, D.O. Son, S.G. Kang, J.M. Kim, H.-H. Lee, C.H. Kim, An efficient scheduling scheme using estimated execution time for heterogeneous computing systems, J. Supercomput. 65 (2013) 886–902.
[46] Ubuntu, Perf, https://www.man7.org/linux/man-pages/man1/perf.1.html.
[47] Nvidia-smi,https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.