Machine Learning-Driven Energy-Efficient Load Balancing for Real-Time Heterogeneous Systems

Taha Abdelaziz Rahmani *, Ghalem Belalem, Sidi Ahmed Mahmoudi, Omar Rafik Merad Boudia

Computer Science department, LIO laboratory, University of Oran1, Ahmed Ben Bella, Oran (Algeria)

Computer Science department, LIO laboratory, University of Oran1, Ahmed Ben Bella, Oran (Algeria)

Computer Science department, Faculty of Engineering, University of Mons, Mons (Belgium)

Computer Science department, LIO laboratory, University of Oran1, Ahmed Ben Bella, Oran (Algeria)

Abstract

Load balancing plays a critical role in ensuring system stability and optimal performance, and as such, it has been a subject of extensive research across diverse computing domains. Particularly, in heterogeneous systems. Such systems integrate various computing devices with distinct architectures and computational power. Each of these devices is designed to execute a particular type of submitted workload with diverse and immediate requirements.

This research introduces a novel load balancing approach that utilizes machine learning to monitor the load distribution of heterogeneous systems in real-time. It estimates the load of the system's devices according to our proposed definition of the load. This definition provided us with an accurate measurement of the actual load, allowing our approach to detect imbalances and locate underloaded devices. Subsequently, our approach assigns applications to those underloaded devices to achieve load balance.

We conducted a comparative analysis between our proposed approach and various established load balancing methods from diverse computing domains. This evaluation included three critical criteria: load balancing, workload execution time, and energy consumption. Additionally, we introduced a novel metric called "IMBALANCE", which quantifies the difference in load distribution among the system devices.

Each of the load balancing approaches was implemented in a real-time heterogeneous system consisting of a master host and three worker servers, each equipped with a Central Processing Unit and Graphical Processing Unit. To ensure a comprehensive validation of our approach, we conducted experiments in three distinct scenarios, each involving different types of workloads with varying levels of computing intensity.

The experimental results consistently demonstrate significant performance improvements compared to alternative approaches across all experimental scenarios. Our proposed approach excels in achieving minimal system imbalance, resulting in superior load balancing. This accomplishment led to reduced execution times for the workloads and minimized device idle periods, ultimately contributing to enhanced energy efficiency.

Keywords: Execution time prediction, Heterogeneous Cluster, Imbalance, Load Balancing, Energy, Pycaret,

* Corresponding author. *E-mail address:* rahmani.taha@edu.univ-oran1.dz

1. Introduction

Modern computing systems are increasingly using GPUs (Graphics Processing Units) to accelerate a wide range of tasks. The primary goal of integrating GPUs is to leverage their high parallel processing power for computationally intensive tasks. Meanwhile, the CPUs (Central Processing Unit) execute the general-purpose and sequential tasks. By combining the strengths of both devices in a heterogeneous architecture, computing systems can achieve significant performance and energy efficiency.

The performance of Heterogeneous systems depends on efficient application scheduling, which involves leveraging each computing device where it excels. However, in most cases, the GPUs execute applications faster than the CPUs, which leads to an uneven distribution of computing load between the system's devices. This load imbalance results in poor performances, including longer response time, low throughput, and waste of computing power and energy.

Load balancing in heterogeneous systems is a complex task for programmers. Its complexity arises from several factors, including the dynamic nature of workloads and varying hardware capabilities where each device is suited for a specific type of computation. Load balancing algorithms must continuously monitor the state of the system, detect workload changes, and adaptively distribute tasks to the most suitable processing unit in real-time. Furthermore, the synchronization between the devices is manually managed at a low level, given that the existing programming frameworks do not provide any scheduling mechanisms.

Defining the load of a device presents another significant challenge. It involves considering the differences in device architectures and the nature of the applications being executed. The applications' nature includes both the processing intensity and the type of computations. Neglecting either factor leads to inaccurate load measurements, resulting in suboptimal performances. For instance, a highly intensive and sequential application performs better on the CPU, despite the GPU having more computational power.

Load balancing has been extensively researched across various domains, given its crucial importance. These domains include Cloud services, Web services, and related fields. Researchers have proposed different load definitions, such as the total bytes serviced by a server [19] or functions based on CPU usage and memory size [20]. However, these definitions solely focus on the utilization of computing resources by the executing applications and often overlook the consideration of applications pending in the queue and the duration of resource usage. While these definitions serve the purpose of evaluating the system's performance, they lack the capability of real-time monitoring for the overall system load.

Real-time load monitoring plays a critical role in achieving efficient load balancing in heterogeneous systems. Through continuous monitoring of the overall system load, it proactively prevents overloading individual devices during the process of mapping applications. Furthermore, real-time load monitoring allows for timely adjustments to the load distribution, eliminating the need for frequent migration of applications from one device to another. This reduces the overhead associated with the application migrations and improves the overall system responsiveness and efficiency.

Defining the load of a device as the total execution time of all pending applications on that device offers a comprehensive approach that takes into account both the devices' characteristics and the applications' nature. Any change in these factors directly affects the execution time of an application, reflecting the varying impact that different applications have on the devices' loads based on the aforementioned factors. However, this definition necessitates knowledge about the execution time of each application, which is often unknown as the applications arrive. Consequently, using this load definition in practice poses considerable challenges and may, in some cases, be unfeasible [13].

In this study, we introduce a novel approach to calculate the load of each device by predicting the execution time of each application. By leveraging predicted execution times, we can overcome the challenge of lacking prior knowledge about the actual execution times of applications.

Predicting the execution time can be approached using two types of methods: traditional methods and machine learning-based methods. Traditional methods involve manual calculations and generally offer less accurate and reliable results compared to machine learning-based approaches [16]. The latter, leveraging the power of machine learning algorithms, can better analyze and model the characteristics of the applications and devices, leading to more precise execution time predictions for CPU and GPU platforms.

In this paper, we present a novel scheduling approach that leverages machine learning techniques to address load imbalances in computing devices. Our approach monitors the real-time loads of devices and effectively maps applications to underloaded devices, thereby preventing overloading and underutilization. To quantify the system's imbalance, we introduce a metric based on our load definition, which serves to evaluate the system's performance.

Furthermore, we adapt several widely used load balancing methods from various domains, such as Cloud and web services, into the context of heterogeneous systems. Through a comparative study, we demonstrate the effectiveness of our approach in achieving improved performance compared to existing methods.

The remainder of the paper is organized as follows: Section 2 provides a background on the related domains; section 3 is an overview of related works. The load balancing problem is discussed in Section 4. Section 5 introduces our proposed load balancing scheduler. In Section 6 and Section 7, we construct the devices' datasets, which are subsequently used in Section 8 for training the execution time prediction models. The experimental results are reported in Section 9. Finally, in Section 10 we summarize our work and discuss potential enhancements for future research.

2. Background

Research on CPU-GPU heterogeneous systems has gained notable interest across diverse domains due to the increasing use of GPUs in various computing environments. This increasing popularity is attributed to the recent advancements in improving GPU programmability and accessibility. The concurrent utilization of CPUs and GPUs for general-purpose computing has demonstrated significant advantages.

2.1. Heterogeneous Systems Programming Model

Applications designed for heterogeneous systems adopt a host-kernel model, comprising two main components: the host and the kernel [34].

The kernel represents the parallel part of the application, implementing computational operations. It can be executed across various computing devices, such as CPUs and GPUs.

On the other hand, the host represents the sequential aspect of the application. Its primary role involves managing the execution of kernels within the system, ensuring proper coordination and synchronization among devices. The host is executed exclusively on the CPU, serving as the central unit for orchestrating parallel kernel computations and handling other sequential tasks such as memory allocations and data copying as needed.

2.2. OpenCL

Various programming frameworks, including OpenCL [15], have been developed specifically for programming heterogeneous systems.

OpenCL is a programming framework for heterogeneous systems, supported by the Khronos Group [15]. It allows parallel programming of modern processors such as GPUs and many-core CPUs [2].

In OpenCL's execution model, the kernel code is executed by launching multiple threads, each known as a work-item, representing an instance of the kernel. These work-items are organized into groups referred to as work-groups. Each work-group runs on a single computing unit within the system. OpenCL provides the ability to execute work-groups of a single kernel independently. Each work-group can be executed on a different device than other work-groups.

2.3. Scheduling In Heterogeneous Systems

In a heterogeneous system, applications can be scheduled using two types of approaches: data-parallel and task-parallel. Data-parallel approaches allow a single application's kernel to be executed concurrently on multiple computing devices. On the other hand, task-parallel approaches execute each application's kernel on a single device [9].

The selection between data-parallel and task-parallel approaches is crucial for optimizing system performance. This decision depends on assessing the specific workload requirements, the capabilities of the available computing devices, and the architecture of the heterogeneous systems.

In scenarios where applications involve the processing of large volumes of data, adopting a data-parallel approach is more effective [9]. This approach effectively distributes the computational load across multiple devices, alleviating the overload on individual devices. However, data-parallel approaches introduce complexities related to data partitioning and distribution, which introduce an additional overhead. To optimize data partitioning, an additional mechanism should be implemented to determine the optimal data partitioning. This mechanism must consider the capabilities of system devices and the network speed within the heterogeneous system, making the programmer's task more challenging. Additionally, the speedup achieved through data parallelization should outweigh the cost of data partitioning and the subsequent transfer of data chunks between central memory and device memories via the network.

In contrast, task-parallel approaches minimize data transfer overhead and are simpler to implement since they don't require an additional data splitting mechanism. Furthermore, the data is transferred only at the initiation and conclusion of application execution. Nevertheless, task parallel approaches are not suitable when dealing with large data volumes if individual device resources, including memory and processing power, are insufficient for the application's requirements.

2.4. Load Balancing

Load balancing is a critical process that distributes the workload equally among computing resources. This process becomes particularly significant in heterogeneous systems, where various devices possess diverse capabilities. Ensuring effective load balancing strategies is essential for maintaining system stability and maximizing the efficiency of the entire computing system. By achieving a balanced workload distribution, load balancing optimizes resource utilization, minimizes response time, and enhances overall system performance.

Load balancing can be addressed through two main approaches: static and dynamic [22].

Static load balancing methods distribute tasks without considering the current state of the computing resources. These methods require prior knowledge of the entire system, including details about job submission times and resource requirements of incoming tasks. While static approaches may be simple to implement, they may not adapt effectively to changing workloads and variations in the system's state.

In contrast, dynamic load balancing approaches exhibit a higher level of complexity but offer significant advantages in terms of efficiency and results. These methods make real-time decisions based on the current system state, incorporating various factors such as the type and complexity of jobs, the overall system architecture, and the current load of computing devices. By adapting to changing conditions, dynamic load balancing can better handle varying workloads and improve overall system performance.

Dynamic load balancing algorithms may introduce overhead due to their complexity. However, their ability to adapt to real-time changes in the system makes them well-suited for dynamic and unpredictable workloads. As computing environments become more heterogeneous and dynamic, the use of dynamic load balancing methods becomes increasingly crucial for achieving better resource utilization and performance.

3. Related Works

In this section, we classify the previous studies on heterogeneous systems based on the scheduling approach employed (data-parallel or task-parallel) and their primary objective (optimizing execution time or load balancing).

We will also include relevant studies from other domains, such as web services, to explore potential solutions that can be adapted to address the challenges of load balancing in heterogeneous systems.

3.1. Data-Parallel Approaches

3.1.1. Execution Time Optimization

In [5], the authors presented a heterogeneous implementation of the Principal Component Analysis (PCA) method in a mobile system. The PCA algorithm was split into two primary components: high precision instructions and computation-intensive instructions. The CPU was assigned the high precision instructions, taking advantage of its proficiency in precision computing tasks. Meanwhile, the GPU was utilized to handle the computation-intensive instructions, leveraging its ability to accelerate parallel processing tasks efficiently.

"SSMART" [12] is a scheduling scheme for heterogeneous systems. Its primary objective is selecting the most appropriate device for task execution. It dynamically evaluates factors such as data transfer time, estimated task execution time, and system workload to select the optimal processing unit for each task. To estimate task execution time, "SSMART" uses statistical data from the history of task executions. The tasks are then allocated to the processing unit with the fastest execution time.

3.1.2. Load Balancing

In [1], the authors investigate the energy efficiency of load balancing for data-parallel applications in heterogeneous systems. The research aims to optimize load balancing techniques to minimize energy consumption in these systems. They evaluate three load balancing approaches for data-parallel applications: static, dynamic, and H-guided. The static load balancing algorithm divides the total workload into a number of workloads equal to the number of devices in the system. Each device is then assigned a workload portion with a size proportional to its computational speed. The computational speed of a device is defined as the amount of work that the device can complete within a unit of time. In contrast, the dynamic algorithm divides the total workload into small, equally sized packages, creating more packages than the number of available devices. Each device is initially assigned one package to process. When a device completes the execution of its assigned package, it is then assigned the next package in line. However, if a device is idle and there are no packages in line, it steals packages from the overloaded devices. On the other hand, the H-guided method shares the same algorithm as the dynamic method but employs a different approach to package sizes. Unlike the dynamic approach with equal-sized packages, the guided algorithm reduces package size as the number of remaining work-groups decreases.

The study in reference [9] introduces "Sigmoid", a scheduler designed for balancing the computing load of a single kernel on the computing devices of a heterogeneous system. This approach assigns work groups, referred to as packages, to computing devices in proportion to their computing powers. Initially, different package sizes are distributed across the devices. As the kernel execution advances, the size of these packages gradually decreases, resulting in more precise load balancing and optimized performance.

3.2. Task-parallel Approaches

3.2.1. Execution Time Optimization

In the study [4], the authors present two distinct scheduling schemes for image processing in heterogeneous systems:

In individual image processing, the computing device selection is determined by two factors: the task's complexity and its parallel computational requirements. To quantify these factors, a complexity metric was introduced. The authors set a predefined threshold to classify applications as complex or non-complex based on their complexity factor. Tasks with complexity values below the threshold are assigned to the CPU, while complex tasks surpassing the threshold are offloaded to the GPU.

As for multiple image processing, the authors employ both static and dynamic scheduling approaches to allocate tasks to the CPU and GPU. In the static approach, tasks are allocated in a first-come, first-served manner, following their order of arrival. In contrast, the dynamic approach assigns tasks to the device having the minimal estimated execution duration. This estimation is made using a model based on historical task execution times.

In the cloud computing context, [6] introduces "KubeSCRTP", a machine learning-based scheduler for Kubernetes. The primary goal of "KubeSCRTP" is to optimize resource utilization and minimize execution time by assigning Docker applications to the appropriate devices, either CPU or GPU. The scheduler uses a machine learning model to classify applications as 'fast execution' or 'slow execution'. Applications classified as 'fast execution' are allocated to the CPU, while those classified as 'slow execution' are offloaded to the GPU.

The paper [21] introduces a scheduling approach for executing OpenCL programs on CPU/GPU heterogeneous platforms. This approach selects the most suitable device for each program by considering program characteristics and underlying hardware. Program characteristics include static features such as the number of instructions, as well as dynamic runtime features such as input data size.

The paper [23] proposes an efficient scheduling scheme for heterogeneous systems. It utilizes estimated execution times of applications to optimize task allocation across the system's devices. The execution time of applications is estimated using their historical execution data. Applications are then mapped to their suitable devices that have minimal estimated execution times.

The research paper [33] introduces an automated approach for optimizing the allocation of applications to CPU and GPU in heterogeneous computing environments. This dynamic allocation process is based on a Machine learning model that predicts the speedup achieved by executing OpenCL applications on the GPU compared to the CPU. Leveraging these predictions, the scheduler assigns each application to the device (CPU or GPU) that is expected to execute the applications faster.

3.2.2. Load Balancing

"E-OSched" [10] is a load balancing scheme that distributes a job pool's workload in proportion to the system's computing resources. Before scheduling, the computational requirements of all applications in the job pool are determined. Applications are then sorted based on processing requirements, with the shortest-sized jobs given priority. "E-OSched" assigns applications with the highest processing requirements to the GPU, while those with lower requirements are assigned to the CPU.

In [3], "Troodon" is an improved version of "E-OSched" [10]. It uses machine learning to classify applications into 'CPU-suitable' and 'GPU-suitable' pools, based on a device suitability model. Additionally, a machine learning speedup predictor estimates the speedup of each application. With the predicted speedup, "Troodon" sorts the 'CPU-suitable pool' in descending order and the 'GPU-suitable pool' in ascending order. These pools are then combined into a single pool, with CPU-suitable applications at the top and GPU-suitable applications at the bottom. Applications with higher predicted CPU speedup are assigned to the CPU, while those with higher predicted GPU speedup are mapped to the GPU. Before scheduling, "Troodon" calculates the estimated load of each device. If a device can't reach its estimated load with suitable applications, non-suitable applications are mapped to that device.

The authors in [11] introduced a device suitability classifier for a multi-CPU/GPU single node heterogeneous system. Using machine learning techniques, this classifier predicts the most suitable device for executing OpenCL applications, focusing on achieving the fastest execution time. The scheduler then schedules and executes the application on the device with the best predicted performance.

In [14], the authors propose "Hbalancer", a machine learning-based load balancer for real-time CPU-GPU heterogeneous systems. It optimizes resource utilization and reduces the system's load imbalance by evenly distributing the workload across the system devices. To achieve this, "Hbalancer" uses machine learning to predict application execution times on each device. "Hbalancer" then calculates the load of each device based on the applications in their queues and selects the device that minimizes load imbalance for each submitted application.

In the context of cluster-based web servers, the study in [13] compares the performance of common load balancing approaches, including Round Robin, Weighted Round Robin, and Least Connections. The Round Robin approach evenly distributes tasks among available devices in a cyclical manner. When new tasks arrive, they are allocated to the next device in rotation. On the other hand, The Weighted Round Robin (WRR) is an extension of the Round Robin algorithm. In WRR, servers are assigned weights based on their processing capabilities. Servers with higher weights receive more requests, allowing them to handle larger workloads. Meanwhile, the Least Connections algorithm assigns incoming requests to the server with the fewest active connections. The objective of this study is to evaluate the performance and efficiency of these techniques in distributing user requests among multiple server nodes.

The research [24] introduces a scheduling method called "Priority-Weighted Round Robin", for Healthcare Monitoring Systems. It assigns priorities to data in order to ensure that the transmission of urgent data is prioritized. This approach employs Weighted Round Robin scheduling to distribute available bandwidth among the devices.

The research in [25] provides an evaluation of the Weighted Round Robin (WRR) load balancing technique for cloud web services. The authors demonstrate the impact of different weight assignments on load distribution and system performance by adjusting weights according to server specifications.

The paper [26] discusses the experimental setup for evaluating popular Load balancing algorithms in a virtual cloud environment. Multiple algorithms are tested, including, Round Robin, Weighted Round Robin, and Least Connections. The primary goal of this study is to evaluate and compare the performance of these algorithms in a simulated cloud environment.

3.3. Discussion

The studies reviewed addressed the scheduling problem in heterogeneous systems. They investigate different approaches to managing task distribution across diverse computing devices, with the goal of optimizing system performance and resource utilization.

Data-parallel scheduling approaches are utilized when a single device cannot execute all the work-groups of an application kernel concurrently. However, these approaches require sufficiently high computational requirements in executed applications to compensate for the overhead involved in kernel partitioning and data transfer between the CPU and other devices.

In contrast, for applications with lower computational intensity, task-parallel approaches prove more appropriate [9]. These approaches enable multiple application kernels to run concurrently, with each device executing all the work-groups of a kernel. Task-parallel strategies offer the advantage of executing a greater number of user applications compared to data-parallel approaches.

The presented task-parallel scheduling approaches have tackled the scheduling problem with diverse objectives. Some prioritize optimizing the execution time of individual applications, while others focus on achieving an equitable distribution of tasks across the devices.

In the studies presented in [4], [6], [11], [21], [23], and [33] the authors prioritize assigning applications to the most suitable devices to minimize their execution times without explicitly considering the load balance within the system. While this device suitability approach can achieve load balancing if the workload is balanced, real workloads often tend to be imbalanced. A workload is considered balanced when the loads of applications suitable for each device are equal. Consequently, this approach leads to overloading the powerful devices, causing an imbalance in the system's load, which results in suboptimal performances in resource utilization and response time. The response time of an application is defined as the duration from the submission of the application to the reception of its results. This duration includes data transfer time, queuing time, and execution time.

In contrast, the schedulers presented in [10] and [3] adopt load balancing approaches where all applications are initially submitted to a job pool, disregarding the order of submission. These approaches assume that the total processing requirements of the job pool are known in advance to estimate the load of each device. However, these schedulers do not consider the architectural differences among devices of the same type. Specifically, the GPUs in the system are assumed to have identical architectures, with the only difference being their computing power measured in flops. This simplification cannot accurately reflect the varying characteristics of the GPUs.

In real-time systems, the dynamic and unpredictable nature of application submissions poses a challenge. Users continuously submit applications at various times, making it impossible to determine the total computing requirements of all applications beforehand. Therefore, previous solutions that rely on pre-calculated workload information are not suitable for real-time systems.

The works [13], [24], [25], and [26] encompass various load balancing methods from different domains. Among these, [13] and [26] implement the "Least Connections" method as a dynamic approach, while [24] and [25] utilize the "Weighted Round-Robin" (WRR) and "Round-Robin" (RR) methods as static approaches. However, in heterogeneous systems, static methods may not perform optimally as they lack adaptability to changing workloads. As for the "Least Connections" method, it does not consider the load of individual requests, assuming that all requests have the same size.

In [14], the authors propose a dynamic load balancing approach for heterogeneous systems. It assigns applications to devices while considering the current state of the system and the workload imbalances. However, the “HBalancer” approach runs in a single node CPU-GPU heterogeneous system. It might not directly suit other types of heterogeneous systems with distinct architectures.

In this work, we propose a scheduling scheme that supports multi-node CPU-GPU heterogeneous systems. Our approach leverages machine learning models to calculate the load of each computing device at run-time using the predicted execution time of applications. Our scheduler dynamically adjusts the assignment of applications at runtime by assigning applications to idle devices. Moreover, we will adapt and compare different load balancing schemes from diverse domains to validate the performance of our proposed method. Furthermore, we introduce a metric to assess the imbalance in workload distribution among the system's devices.

4. Load Balancing Problem

Load balancing is a computational optimization technique used to distribute processing tasks evenly across multiple computing resources. This approach aims to optimize the utilization of resources and minimize the system's response time, resulting in improved processing efficiency.

In real-time heterogeneous systems, the computing power of devices can vary significantly, and applications have varying types and computing intensities. As a result, load balancing in such systems becomes more complex. Effective load balancing methods should be capable of monitoring the system's state, adapting to workload changes, and considering devices' load state. By using this information, these methods can make real-time decisions to select the best devices for executing applications.

Providing a correct definition of a device load is crucial for the effective performance of load balancing methods. It should integrate various factors that influence application execution on the devices, such as device computing capability, device architecture, application characteristics, and application type. Neglecting any of these factors can result in incorrect load estimation and subsequently lead to suboptimal performance of the load balancing method.

We define the load of a device as "The time required to execute all pending applications assigned to the device". This definition takes into account the aforementioned factors. It considers that any changes in these factors will be reflected in the application execution time, thereby influencing the device load. By considering the execution time of assigned applications, our measure provides a reliable and accurate assessment of device load in heterogeneous systems.

In order to evaluate the system's performance, we define a metric called "IMBALANCE". This metric is defined as “The total difference between the loads of all devices in the system”. Details regarding the calculation of the "IMBALANCE" metric will be provided in the subsequent sections.

Fig. 1 illustrates two heterogeneous systems, each comprising multiple CPUs and GPUs. In both systems, each computing device is allocated a distinct set of applications.

In *Fig. 1 (a)*, we observe a well-balanced system with equal queue sizes for all devices. This indicates a fair distribution of workload among the devices, leading to synchronized completion times for the execution of the assigned applications.

In contrast, *Fig. 1(b)* illustrates an imbalanced system where some devices are overloaded, while others are underloaded. Consequently, the underloaded devices finish their executions earlier and remain inactive, resulting in wasteful energy consumption.

To address this load imbalance, a load balancing strategy can be employed to redistribute applications from overloaded devices to underloaded ones.

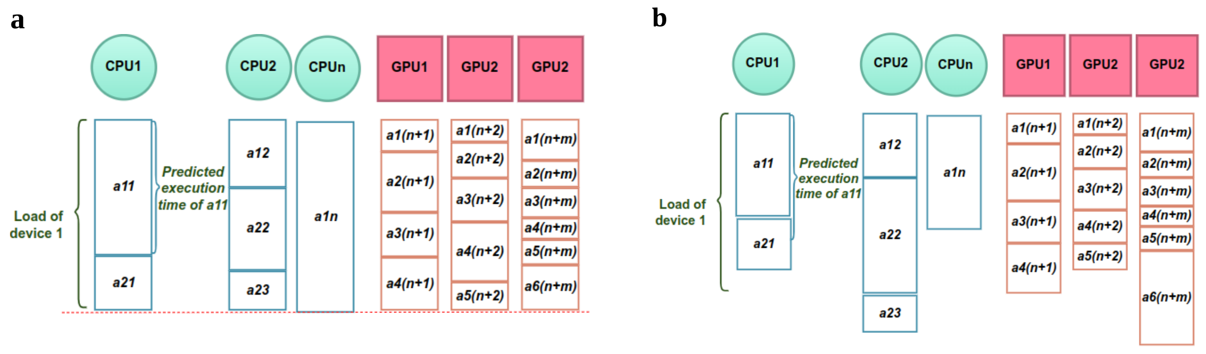


Fig. 1 (a) Balanced system; **(b)** Imbalanced system.

5. Least Loaded Device Scheduler (LLD)

In this section, we will present an overview of our proposed scheduler, LLD, which includes details about its system architecture, scheduling algorithms, and a mathematical model that describes its functionality and enables the calculation of devices’ loads and the system imbalance.

5.1. System Architecture

In real-time heterogeneous systems, user applications are continuously submitted. The application scheduler must rapidly allocate the applications to devices for execution to minimize response time. Each application has a limited number of possible schedulings equal to the number of available devices. Our scheduler continuously monitors all the devices' loads. It then selects the best scheduling that balances the load between the devices by assigning applications to the underloaded device.

Fig. 2 illustrates the global architecture of our LLD load balancing scheduler. The scheduler operates through a six-step workflow:

1. Application submission: Users submit their applications to the scheduler queue. The scheduler sorts the submitted applications in ascending order of the submission time.
2. Feature extraction: The scheduler retrieves the first application kernel from the queue and sends it to the Features extractor module. The Features extractor extracts the code features of the application and sends them to the Execution time predictor.
3. Execution time prediction: The Execution time predictor uses pre-trained machine learning models to predict the execution time of the application on all available devices. The predicted execution times are then sent back to the scheduler.
4. Device selection: The scheduler selects the most underloaded device based on the minimal calculated device loads.
5. Application submission: The application is submitted to the queue of the selected device for execution.
6. Load update: The scheduler updates the load of the selected device by adding the predicted execution time of the application to its load.

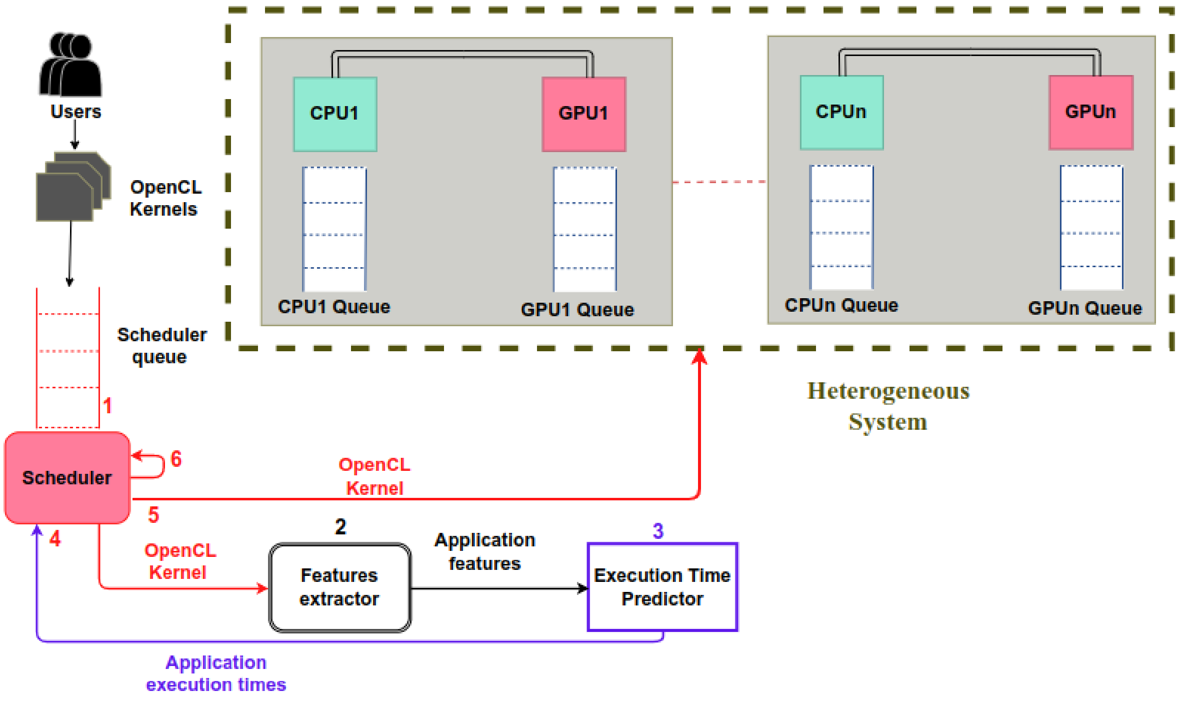


Fig. 2 The system's architecture

5.2. Scheduling Algorithms

Algorithm 1 presents the load balancing algorithm LLD, while *Algorithm 2* presents the execution time prediction model.

Algorithm 1: Load balancing scheduling

Input: 'A' Applications submitted by users

Output: Computing device for each application

```
START
1   sched_queue ← A
2   sort(sched_queue)
3   While sched_queue not empty:
4       application ← sched_queue (0)
5       $F_i$ ← feature_extractor(application)
6       Time ← Execution_time_predictor($F_i$)
7       min_load ← 0
8       For every device j:
9           if $Load_j < Load_{min\_load}$:
10          min_load ← j
11      End For
12      computing_device $_{min\_load}$ ← application
13      $Load_{min\_load}$ ← $Load_{min\_load} + Time_{min\_load}$
14  End While
END
```

Algorithm 2: Execution time prediction

Input: F //*Kernel code features*

Output: Predicted time on all devices

```
START
1   For every device j:
2       $Time_j$ ← prediction_model($F_i$)
    End For
END
```

The LLD scheduler algorithm takes a set of applications as input and generates a scheduling plan for each application as output.

In *Algorithm 1*, the submitted applications are arranged in the scheduler queue in line 1. Then, the scheduler sorts the applications in its queue in ascending order of the submission time in line 2.

in line 4, the scheduler retrieves the application at the top of its queue and passes it as an argument when calling the "feature_extractor()" module in line 5. The "feature_extractor()" function returns a set of features.

The "Execution_Time_predictor()" function in line 6, takes the extracted features and predicts the execution time of the application on all available devices in the system, as indicated in *Algorithm 2* line 2.

From line 8 to 10, the scheduler algorithm selects the device with the minimal load, and the application is then mapped to this device in line 12.

In line 13, the load of the selected device is updated by adding the predicted execution time of the application on the selected device to its current load.

5.3. Mathematical Model

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

Table 1. Notations

Notation	**Description**
A	Submitted OpenCL applications
S	Heterogeneous system
n	Number of the system's devices
sched_queue	Scheduler's queue
Time	List of predicted execution time of an application on all the system's devices
Time_j	Predicted execution time of an application on the node 'j'
Time_{ij}	Predicted execution time of the application number 'i' mapped to the node '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'
IMBALANCE	The system's imbalance
$\text{Energy}_{\text{active } j}$	The energy consumed by node 'j' per second when executing tasks
$\text{Energy}_{\text{idle } j}$	The energy consumed by node 'j' per second when idle

We define the Load of a device 'j' "Load_j" as the sum of the predicted execution times "Time_{ij}" of all the applications mapped to the device 'j'. "Load_j" is defined as follows:

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

To measure the load balancing and energy performances, we introduce the "IMBALANCE" and "Energy" metrics.

When a system is perfectly balanced:

$$\begin{cases} Load_1 = Load_2 \\ \quad\cdots \\ Load_1 = Load_n \\ \quad\cdots \\ Load_{n-1} = Load_n \end{cases} \Leftrightarrow \begin{cases} Load_1 - Load_2 = 0 \\ \quad\cdots \\ Load_1 - Load_n = 0 \\ \quad\cdots \\ Load_{n-1} - Load_n = 0 \end{cases} \tag{2}$$

We generalize the equations in (2) into a single equation:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Load_i - Load_j = 0 \tag{3}$$

When two devices 'i' and 'j' are imbalanced:

$$\begin{cases} Load_i < Load_j \Leftrightarrow Load_i - Load_j < 0 \\ \qquad\qquad or \\ Load_i > Load_j \Leftrightarrow Load_i - Load_j > 0 \end{cases} \Leftrightarrow \quad \left|Load_i - Load_j\right| > 0 \tag{4}$$

"IMBALANCE" is defined as in *equation 5*:

$$IMBALANCE = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left| Load_i - Load_j \right| \tag{5}$$

The energy consumed by a device during the experiment is calculated based on both its active and idle energy consumption. Throughout the experiment, each device operates in one of two states: idle, where it functions without performing computations, or active, where it executes computational tasks.

The active energy of a device 'j' refers to the cumulative energy consumed by the device while executing computational tasks, it is equal to the total active time of the device 'j' multiplied by its active energy consumption per second:

$$TotalEnergy_{active\ j} = Time_{active\ j} * Energy_{active\ j} \tag{6}$$

Likewise, the idle energy refers to the total energy consumed when the device is idle:

$$TotalEnergy_{idle\ j} = Time_{idle\ j} * Energy_{idle\ j} \tag{7}$$

The total energy of a device is then defined as:

$$TotalEnergy_j = TotalEnergy_{active\ j} + Total_Energy_{idle\ j} \tag{8}$$

Consequently, the total energy consumed by all the system's devices is the cumulative energy consumed by all the devices in the system (*equation 9*):

$$TotalEnergy = \sum_{j\,=\,1}^{n} TotalEnergy_j \tag{9}$$

6. Dataset Creation

The process of training prediction models for deployment includes three main steps: dataset preparation, model training, and model validation.

6.1. Dataset creation

Table 2. OpenCL's code features

No.	Features	Extraction method	No.	Features	Extraction method
1	Data Size	Regular Expressions	10	Subtractions	LLVM pass
2	Return instructions	LLVM pass	11	Function Calls	LLVM pass
3	Control Instructions	LLVM pass	12	Functions	LLVM pass
4	Memory Loads	LLVM pass	13	Blocks of Instructions	LLVM pass
5	Memory Stores	LLVM pass	14	Integer Operations	Regular Expressions
6	Multiplications	LLVM pass	15	Float Operations	Regular Expressions
7	Divisions	LLVM pass	16	Total Instructions	LLVM pass
8	Conditions	LLVM pass	17	Loop Operations	LLVM pass
9	Additions	LLVM pass	18	Loops	LLVM pass

The dataset creation process involved two main steps: feature extraction and application execution.

In this process, we used the Polybench suite [7], a collection of OpenCL benchmarks covering diverse application domains, including pattern recognition and mathematical computations. This suite ensured that the dataset represented various types of real-world OpenCL applications. The specific Polybench applications along with our own developed Matrix transpose application are listed in Table 3.

Table 3. OpenCL applications

	Application		**Application**		**Application**		**Application**
1	2mm	6	atax	11	doitgen	16	jacobi-2D
2	2D-Convolution	7	bicg	12	gemm	17	mvt
3	3mm	8	correlation	13	gemver	18	syr2k
4	3D-Convolution	9	covariance	14	gesummv	19	syrk
5	Adi	10	fdtd-2d	15	jacobi-1D	20	transpose

6.2. Features Extraction

This process involved extracting a set of 18 OpenCL kernel code features from each application, capturing various characteristics that could impact its execution time. These features are listed in *Table 2*.

A kernel code feature extractor was implemented using the Low-Level Virtual Machine (LLVM) compiler and regular expressions to extract code features from OpenCL kernels.

The LLVM compiler offers a readable intermediate representation (IR) of programs, which the feature extractor uses to extract 15 features. The remaining three features are extracted using regular expressions. *Fig. 3* illustrates the features extraction process.

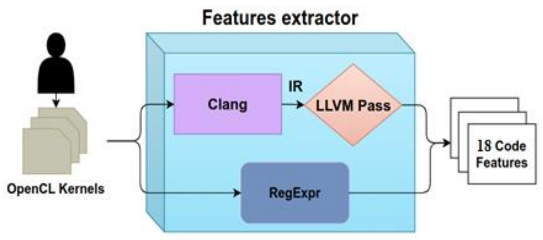


Fig. 3 OpenCL code features extractor

Each kernel is first compiled using the Clang front-end compiler to verify its code correctness. This compilation step generates an LLVM intermediate representation (IR). From this IR, we apply an LLVM pass to extract 15 specific code features. Three additional features are extracted using regular expressions. Through this process, we obtain a total of 18 code features, as listed in *Table 2*.

6.3. OpenCL Applications Execution

For each computing device in the heterogeneous cluster, we executed each application 2,000 times with varying input data sizes. This resulted in a total of 40,000 distinct executions of applications in each device. The resulting dataset consists of 40,000 rows and 18 columns, representing the code features of the applications and their corresponding execution times on the respective device. The execution time is set as the target variable for prediction. This process was repeated for each available computing device within the heterogeneous cluster.

The training of prediction models will be performed on a separate machine, and the resulting models will later be deployed alongside the scheduler.

7. Dataset Preprocessing

Data preprocessing is a crucial step in machine learning that involves transforming the raw data into a format that can be used by machine learning models. It is used to select only the most relevant information from the dataset for training.

PyCaret [18] is a Python library that provides a simple and intuitive interface for machine learning tasks. It was used to preprocess the dataset, train, validate, and deploy the execution time prediction models.

7.1. PyCaret

PyCaret is an open-source machine learning library developed in Python that aims to simplify the process of developing machine learning models [18]. It provides a range of functionalities for automatic model creation, cross-validation, metric evaluation, model comparison, hyperparameter tuning, and analysis of trained models. PyCaret also provides pre-processing capabilities such as removing multicollinearity and performing features selection. These operations can be easily performed by simply calling the specific functions within the PyCaret framework. Overall, PyCaret helps users to quickly build and evaluate machine learning models while reducing the need for manual coding and configuration.

Pycaret workflow is depicted in *Fig.4*.

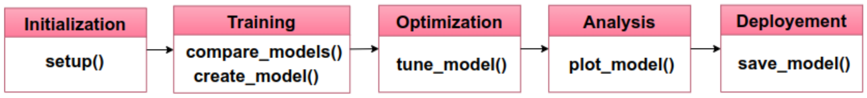


Fig. 4 Pycaret workflow

The following functions are used to perform the different steps in the workflow [18]:

setup (): Prepares the transformation pipeline using the parameters provided in the function.

compare_models (): Trains, evaluates, and compares several machine learning algorithms using cross-validation.

create_model (): Trains and evaluates a specific machine learning algorithm using cross-validation.

tune_model (): Tunes the hyperparameters of a trained model.

plot_model (): Allows to visualize the performance of a trained model. It also performs recursive features selection and evaluation of features importance in contributing to the model's performance.

save_model (): Saves the trained model as a pickle file for later use.

7.2. Features Selection

Feature selection is a fundamental step in machine learning that aims to reduce the dimensionality of the dataset. This involves identifying and extracting the most discriminative and relevant features that impact the precision of the models while excluding the irrelevant and redundant ones [35].

Using a reduced dataset for training a machine learning model offers several advantages. First, it decreases the training time, which enhances computational efficiency. Second, it improves the interpretability of the models by focusing on an informative set of features. This enables the models to capture the underlying patterns and relationships within the data, providing more accurate predictions. Third, it aids in generalizing the model by reducing the risk of overfitting. This is because the model becomes less susceptible to memorizing noise in the training data.

In summary, the process of feature selection enhances the predictive accuracy and the model robustness. In our study, we use three different features selection methods to ensure that the best features are selected:

- Correlation Analysis: By employing correlation analysis, we identify correlated features to validate the upcoming feature selection decisions.
- Feature Ranking via Decision Tree Analysis: During this stage, we assess the impact of each feature on model performance and rank them accordingly.
- Recursive Feature Selection: We employ recursive feature selection to determine the ideal number of features needed to reach the best model performance.

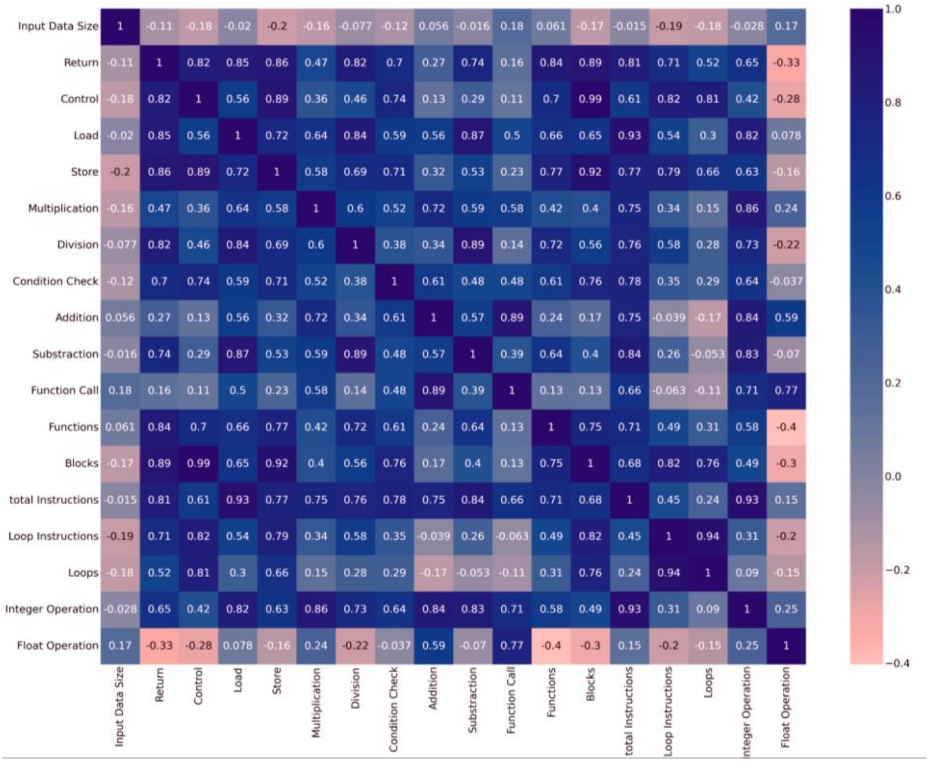
7.2.1. Correlation analysis

Correlation analysis is a statistical method that can be used to examine the dependencies between the different features of a dataset. It quantifies the relationship between all the pairs of features and can be used to identify highly correlated features. When features are highly correlated, they provide similar information, which means they are redundant [3]. Including both correlated features in a model can lead to reduced accuracy, overfitting, and increased training time. Overfitting occurs when a model is excessively tuned to the training data, resulting in a limited ability to generalize to new data.

By removing correlated features, the model will be less likely to learn patterns that are specific to the training data, preventing overfitting and improving the model's accuracy.

Fig. 5 displays the correlation matrix of the features listed in *Table 2*. The highly correlated pairs of features with correlation values greater than 0.9 are:

- (“Control”, “Blocks”)
- (“Load”, “Total Instructions”)
- (“Store”, “Blocks”)
- (“Total Instructions”, “Integer Operations”)
- (“Loop Instructions”, “Loops”)



	Input Data Size	Return	Control	Load	Store	Multiplication	Division	Condition Check	Addition	Substraction	Function Call	Functions	Blocks	total Instructions	Loop Instructions	Loops	Integer Operation	Float Operation
Input Data Size	1	-0.11	-0.18	-0.02	-0.2	-0.16	-0.077	-0.12	0.056	-0.016	0.18	0.061	-0.17	-0.015	-0.19	-0.18	-0.028	0.17
Return	-0.11	1	0.82	0.85	0.86	0.47	0.82	0.7	0.27	0.74	0.16	0.84	0.89	0.81	0.71	0.52	0.65	-0.33
Control	-0.18	0.82	1	0.56	0.89	0.36	0.46	0.74	0.13	0.29	0.11	0.7	0.99	0.61	0.82	0.81	0.42	-0.28
Load	-0.02	0.85	0.56	1	0.72	0.64	0.84	0.59	0.56	0.87	0.5	0.66	0.65	0.93	0.54	0.3	0.82	0.078
Store	-0.2	0.86	0.89	0.72	1	0.58	0.69	0.71	0.32	0.53	0.23	0.77	0.92	0.77	0.79	0.66	0.63	-0.16
Multiplication	-0.16	0.47	0.36	0.64	0.58	1	0.6	0.52	0.72	0.59	0.58	0.42	0.4	0.75	0.34	0.15	0.86	0.24
Division	-0.077	0.82	0.46	0.84	0.69	0.6	1	0.38	0.34	0.89	0.14	0.72	0.56	0.76	0.58	0.28	0.73	-0.22
Condition Check	-0.12	0.7	0.74	0.59	0.71	0.52	0.38	1	0.61	0.48	0.48	0.61	0.76	0.78	0.35	0.29	0.64	-0.037
Addition	0.056	0.27	0.13	0.56	0.32	0.72	0.34	0.61	1	0.57	0.89	0.24	0.17	0.75	-0.039	-0.17	0.84	0.59
Substraction	-0.016	0.74	0.29	0.87	0.53	0.59	0.89	0.48	0.57	1	0.39	0.64	0.4	0.84	0.26	-0.053	0.83	-0.07
Function Call	0.18	0.16	0.11	0.5	0.23	0.58	0.14	0.48	0.89	0.39	1	0.13	0.13	0.66	-0.063	-0.11	0.71	0.77
Functions	0.061	0.84	0.7	0.66	0.77	0.42	0.72	0.61	0.24	0.64	0.13	1	0.75	0.71	0.49	0.31	0.58	-0.4
Blocks	-0.17	0.89	0.99	0.65	0.92	0.4	0.56	0.76	0.17	0.4	0.13	0.75	1	0.68	0.82	0.76	0.49	-0.3
total Instructions	-0.015	0.81	0.61	0.93	0.77	0.75	0.76	0.78	0.75	0.84	0.66	0.71	0.68	1	0.45	0.24	0.93	0.15
Loop Instructions	-0.19	0.71	0.82	0.54	0.79	0.34	0.58	0.35	-0.039	0.26	-0.063	0.49	0.82	0.45	1	0.94	0.31	-0.2
Loops	-0.18	0.52	0.81	0.3	0.66	0.15	0.28	0.29	-0.17	-0.053	-0.11	0.31	0.76	0.24	0.94	1	0.09	-0.15
Integer Operation	-0.028	0.65	0.42	0.82	0.63	0.86	0.73	0.64	0.84	0.83	0.71	0.58	0.49	0.93	0.31	0.09	1	0.25
Float Operation	0.17	-0.33	-0.28	0.078	-0.16	0.24	-0.22	-0.037	0.59	-0.07	0.77	-0.4	-0.3	0.15	-0.2	-0.15	0.25	1

Fig. 5. Correlation matrix

To decide which feature to remove within each pair, it is necessary to evaluate the importance and impact of each feature on the model's performance. The features that have the least impact on the model's performance are excluded from the dataset.

7.2.2. Decision Tree Features Importance Ranking

The decision tree-based feature importance ranking quantifies each feature’s contribution to the model training process. It assigns ranks to the features based on their importance scores. This ranking helps to identify the most important features within the dataset [35]. The features ranking is reported in *Fig.6*.

7.2.3. Recursive Features Selection

Recursive feature selection (RFS) is the process of selecting features in a dataset by iteratively removing features based on their importance scores. In each iteration, one feature is removed, and the model is retrained using the remaining features. The performance of the model is then evaluated using a metric, such as the coefficient of determination (r^2). This process is repeated until all features have been removed [36]. The goal of recursive feature selection is to identify the most important features that contribute significantly to the model's performance. The results of the RFS are reported in *Fig. 7*.

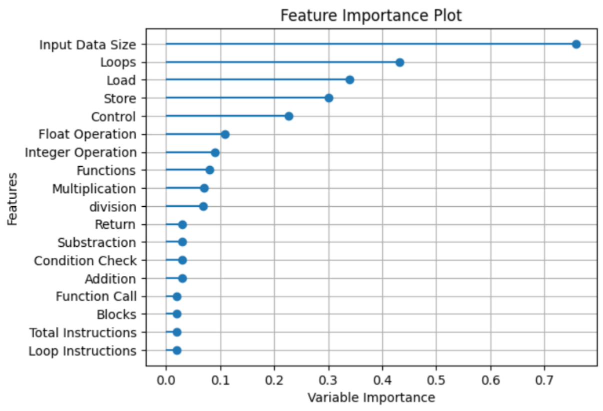


Fig. 6 Features ranking

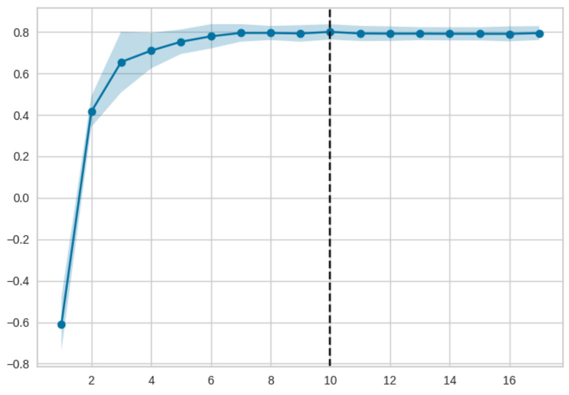


Fig. 7 Recursive features selection

7.3. Selected features

The results of the recursive feature selection process indicate that the top 10 ranked features are sufficient to achieve the highest accuracy. Consequently, the remaining features were excluded. *Table 4* presents the 10 highest-ranked features, as determined by the feature importance ranking process.

Table 4. Selected OpenCL features

No.	Features
1	Input Data Size
2	Loops
3	Load
4	Store
5	Control
6	Float Operation
7	Integer Operation
8	Function
9	Multiplication
10	Division

The correlation analysis results further validate the suitability and optimality of the selected features. The low correlation among these features confirms that they are appropriate for the model.

8. Execution Time Prediction Model Training

After preprocessing the datasets and selecting the relevant features, the resulting datasets are used for training the prediction models.

8.1. Training of execution time model

The prepared datasets were divided into two subsets: the train/validation set and the test set. The train/validation set comprises 90% of the data, while the remaining 10% was assigned to the test set. Within the train/validation set, 90% is used for training, while the remaining 10% serves as a validation subset.

To train the machine learning models, we first initialize the environment for each model by using the setup function. The setup function takes as input the train/validation dataset and the target variable (execution time). It then configures the necessary settings and prepares the data for model training.

To compare the performance of different machine learning models, we used the cross-validation functionality provided by the PyCaret library. This cross-validation technique trains the models on different combinations of 90/10 % data splits of the train/validation subset. The resulting output from cross-validation represents the average cross-validated scores of the models.

The cross-validation results of the three best models are presented in Tables 5, 6, 7, 8, 9, and 10. These tables present various evaluation metrics, including mean squared error (MAE), root mean squared error (RMSE), and R-squared (r^2). For each dataset, we selected the best performing model.

RMSE and MAE metrics evaluate the accuracy of predictive models. They both calculate the difference between predicted and actual values for each prediction but employ different approaches. RMSE exhibits greater sensitivity to extreme values with substantial prediction errors since it squares the errors before averaging them. In contrast, MAE treats all errors equally, regardless of their magnitude. This characteristic renders MAE less susceptible to extreme values, enhancing its robustness and interpretability. On the other hand, the R-squared metric indicates how effectively models represent the data [37].

A higher R-squared value suggests a superior fit, while minimal RMSE and MAE values indicate heightened accuracy.

For each computing device, we selected the best performing models having higher r^2 values and minimal MAE and RMSE values. The selected models are Extra Trees Regressor for CPU1, GPU1 and CPU2 and Gradient Boosting Regressor for GPU2, CPU3 and GPU3.

Table 5. CPU1 execution time prediction model's training

	Model	**MAE**	**RMSE**	**R2**
et	Extra Trees Regressor	0.0062	0.0123	0.9498
rf	Random Forest Regressor	0.0090	0.0132	0.9407
dt	Decision Tree Regressor	0.0069	0.0137	0,9361

Table 6. GPU1 execution time prediction model's training

	Model	**MAE**	**RMSE**	**R2**
et	Extra Trees Regressor	0.0026	0.0054	0.9529
rf	Random Forest Regressor	0.0038	0.0057	0.9443
dt	Decision Tree Regressor	0.0029	0.0060	0,9441

Table 7. CPU2 execution time prediction model's training

	Model	**MAE**	**RMSE**	**R2**
et	Extra Trees Regressor	0.0035	0.0088	0.9612
dt	Decision Tree Regressor	0.0039	0.0098	0.9298
rf	Random Forest Regressor	0.0072	0.0107	0,9236

Table 8. GPU2 execution time prediction model's training

	Model	**MAE**	**RMSE**	**R2**
gbr	Gradient Boosting Regressor	0.0051	0.0067	0.9427
ada	AdaBoost Regressor	0.0056	0.0068	0.9400
Knn	K Neighbors Regressor	0.0056	0.0080	0,9184

Table 9. CPU3 execution time prediction model's training

	Model	**MAE**	**RMSE**	**R2**
gbr	Gradient Boosting Regressor	0.0127	0.0160	0.9339
lighgbm	Light Gradient Boosting Machine	0.0132	0.0166	0.9287
rf	Random Forest Regressor	0.0125	0.0167	0,9282

Table 10. GPU3 execution time prediction model's training

	Model	MAE	RMSE	R2
gbr	Gradient Boosting Regressor	0.0053	0.0073	0.9273
ada	AdaBoost Regressor	0.0056	0.0073	0.9270
lighgbm	Light Gradient Boosting Machine	0.0053	0.0073	0,9224

8.2. Test and Deployment

To enhance the performance of the finalized models, we retrain them using the combined training and validation datasets. This step allows us to fine-tune the models' hyperparameters using the tune_model() function, which automatically searches for the best hyperparameter combinations for each model.

The tuned models are subjected to a final evaluation using the test set, which essentially represents data that the model has never seen during its training or validation phases. This evaluation provides a rigorous assessment of how well the model generalizes to unseen data.

Table 11, Table 12, Table 13, Table 14, Table 15, and *Table 16* present the validation results of the tuned models.

Table 11. CPU1 execution time prediction model's validation

	Model	MAE	RMSE	R2
et	Extra Trees Regressor	0.0053	0.0116	0.9533

Table 12. GPU1 execution time prediction model's validation

	Model	MAE	RMSE	R2
et	Extra Trees Regressor	0.0023	0.0050	0.9684

Table 13. CPU2 execution time prediction model's validation

	Model	MAE	RMSE	R2
et	Extra Trees Regressor	0.0026	0.0054	0.9787

Table 14. GPU2 execution time prediction model's validation

	Model	MAE	RMSE	R2
gbr	Gradient Boosting Regressor	0.0034	0.0062	0.9523

Table 15. CPU3 execution time prediction model's validation

	Model	MAE	RMSE	R2
gbr	Gradient Boosting Regressor	0.0124	0.0159	0.9439

Table 16. GPU3 execution time prediction model's validation

	Model	MAE	RMSE	R2
gbr	Gradient Boosting Regressor	0.0053	0.0073	0.9441

9. Experimental Results

Our proposed scheduler was deployed on a multi-node CPU-GPU heterogeneous cluster to evaluate its effectiveness under different scenarios. The experimental setup enabled the assessment of the scheduler's performance across different workload distributions, providing a deeper understanding of its adaptability and efficiency in real-world environments.

9.1. Experimental Setup

9.1.1. System

The system architecture includes a master host and three worker servers, each server has a CPU and a GPU. The master host is responsible for distributing computing tasks among the available computing resources in the worker servers. The specific characteristics of the cluster can be found in *Table 17*. The characteristics of the computing nodes are listed in *Table 18*.

Table 17. 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
Slave 1	Ubuntu Server 18.04.1 LTS	8 GB	Intel® Core™ i7-6700U	NVIDIA Corporation GM107GL Quadro K620
Slave 2	Ubuntu Server 18.04.1 LTS	8 GB	Intel® Xeon® E3-1225 v6	NVIDIA Corporation GM107GL Quadro P400
Slave 3	Ubuntu Server 18.04.1 LTS	8 GB	Intel® Xeon® E3-1225 v6	NVIDIA Corporation GM107GL Quadro P400

Table 18. Cluster's devices characteristics

Device	Characteristics
Intel® Core™ i5-6300U	2 cores and 4 threads. Basic Frequency = 2.30 GHZ. Turbo frequency = 2.80 GHZ. Cache = 3 MB.
Intel HD Graphics 520 graphics card	24 execution units. 192 shading units. Maximum frequency = 1.1 GHz.
Intel® Core™ i7-6700U	4 cores and 8 threads. Basic Frequency = 3.40 GHz. Turbo frequency = 4.00 GHz. Cache = 8 MB.
NVIDIA Quadro K620	2048MB memory. Frequency = 1058 MHz, Boosted frequency = 1124 MHz,384 cores
Intel® Xeon® E3-1225 v6	4 cores and 4 threads. Basic Frequency = 3.30 GHZ. Turbo frequency = 3.70 GHZ. Cache = 8 MB.
NVIDIA Corporation GM107GL Quadro P400	2048MB memory. Frequency = 1228 MHz, Boosted frequency = 1252 MHz, 256 cores

9.1.2. Workload

In our experiments, we selected a set of 20 benchmark applications listed in Table 3. Each application was executed with five different input data sizes, resulting in a total of 100 jobs. To ensure the reliability of our results, we conducted each experiment 100 times and reported the average values from these repetitions.

9.1.3. Scenarios

To simulate real-world computing scenarios, we employed three distinct workload types. These workloads were categorized based on the computational demands of their applications, with specific consideration given to the size of the input data. The three categories included medium workload, heavy workload, and mixed workload.

Medium workload: Comprises applications with moderate computational requirements. The input data for these applications is relatively small.

Heavy workload: Comprises applications with high computational requirements. The input data for these applications is relatively large.

Mixed workload: Comprises a combination of both medium and heavy workloads, with applications having varying input data sizes, either small or large.

Table 19 presents the application sizes for each workload in the three considered scenarios.

Table 19. Experimentation scenarios

	Workload	Applications Input Data Size (number of elements)
Scenario 1	Medium	[10,000,000 - 100,000,000]
Scenario 2	Heavy	[100,000,000 - 10,000,000,000]
Scenario 3	Mixed	[10,000,000 -10,000,000,000]

9.1.4. Baselines

To validate our proposed approach, we compared it with several scheduling techniques, including some that we adapted from other domains. These techniques are well-established and remain applicable across a range of computing domains.

1. **Round Robin (RR)**
   Also known as Alternative Assignment, it assigns tasks to the devices in a cyclical manner. As new tasks arrive, they are allocated to the next device in rotation. The round robin approach was implemented in the works [13], [24], [26].
2. **Weighted Round Robin (WRR)**
   WRR is an extended version of the round-robin algorithm. Servers are assigned weights according to their processing capabilities. Servers with higher weights receive a greater number of tasks. This approach was implemented in [13], [24],[25],[26].

- **Calculating devices' weights:**
  Each computing device is assigned a weight proportionally to its computing capabilities. The sum of the weight of all devices is equal to one [24].
  In our system, we calculate the weight of each device based on its number of cores and frequency. The weight of a device is determined using *equation 10*. The reported weight values were rounded to two decimal places.

$$weight = \frac{\left(\frac{Number\ of\ Cores}{Total\ Number\ of\ Cores}\right) + \left(\frac{Frequency}{Total\ Frequency}\right)}{2} \tag{10}$$

Table 20. Weight calculation

Device	Model	Cores	Frequency (Ghz)	Weight	Weight/Min_Weight
CPU1	i7-6700	8	4	0.14	1
GPU1	Quadro K620	384	1,124	0.25	2
CPU2	Xeon(R) CPU E3-1225	4	3,70	0.13	1
GPU2	Quadro P400	256	1,252	0.18	1.5
CPU3	Xeon(R) CPU E3-1225	4	3,7	0.13	1
GPU3	Quadro P400	256	1,252	0.18	1.5

Table 20 represents the weight calculation for each device in the system.
According to the weight distribution:

- CPU1, CPU2, and CPU3 each take one application in every round.
- GPU1 takes 2 applications in every round.
- GPU2 and GPU3 each take three applications in every two rounds. in one round, each GPU takes one application, and in the subsequent round, each GPU takes two applications.

3. **Device Suitability (DS)**
   The Device Suitability assigns applications to the devices that execute them faster. It was implemented in the works [4], [6], [11], [21], [23].
4. **Least Connections (LC)**
   In our adapted version of the Least Connections algorithm, applications are assigned to the device with the least number of applications in its queue. This method was implemented in [13], [26].

9.1.5. Metrics

We employed the following metrics to compare the performance of the mentioned methods with our proposed approach across various scenarios:

- **“IMBALANCE”:** This metric represents the total difference in load between all of the system's devices at a specific point in time during the experiment.
- **“Total imbalance”:** This metric represents the cumulative imbalances of the entire experiment, providing an overall measure of the system's load balancing performance. A smaller total imbalance value indicates a more efficient load balancing performance throughout the experiment.
- **“Finishing time”:** Represents the time required to execute the total workload in the system. It indicates how quickly the system can process all the tasks assigned to it.
- **“Device total execution time”:** Represents the time required for each device to finish all applications assigned to it. It helps assess the workload distribution among the devices.
- **“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.
- **“Energy consumption”:** This metric measures the total energy consumed by the devices throughout the experiment. This includes the energy consumed when the devices are idle, as well as the energy consumed when they are performing computing tasks.

9.2. Experimental Results

During the experiments, the overhead time associated with feature extraction and execution time prediction was accounted for. All approaches used the same prediction models to calculate load imbalance.

The results of the experiment are presented in *Table 22*, *Table 23, and Table 24*, which summarize the performance of the different approaches under the three scenarios.

9.2.1. Imbalance

Fig. 8, *Fig. 9, and Fig. 10* depict the "IMBALANCE" curves of the system for the three scenarios. These figures illustrate the system's behavior throughout the entire experiment using the cited approaches. The time measurements for *Fig. 8*, *Fig.* 9, and *Fig.10* were normalized to enable a clear comparison of the behavior exhibited by the five schedulers.

The LLD approach maintains a consistently low level of imbalance, staying close to zero throughout the scheduling of applications for the three scenarios. However, the curves of the other approaches diverge as more applications are scheduled, indicating higher levels of imbalance in comparison to the LLD approach.

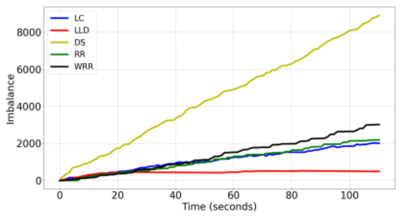


Fig. 8 Imbalance curve for medium workload

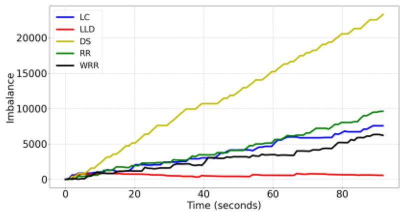


Fig. 9 Imbalance curve for Heavy workload

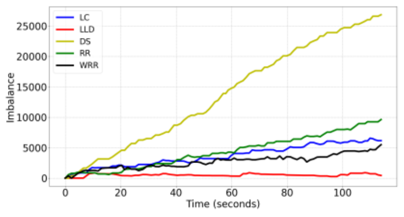


Fig. 10 Imbalance curve for Mixed workload

9.2.2. Total Imbalance

Fig. 11, *Fig.* 12, and *Fig.* 13 depict the total imbalances for the three scenarios. The results indicate that the LLD approach achieves the least total imbalance in all cases, outperforming the other approaches. This suggests that our approach is more effective in achieving balanced load distribution among the devices throughout the experiment.

- According to *Fig. 11*, for a medium workload, “LLD” reduces the total imbalance with 62 % less than Least Connections, 90% less than Device Suitability, 65% less Round Robin, and 68% less than Weighted Round Robin.

- According to *Fig.* 12, for a heavy workload, "LLD" reduces the total imbalance with 85% less than Least Connections, 95% less than Device Suitability, 87% less Round Robin, and 80% less than Weighted Round Robin.
- According to *Fig.* 13, for a mixed workload, "LLD" reduces the total imbalance with 78% less than Least Connections, 94% less than Device Suitability, 86% less Round Robin, and 76% less than Weighted Round Robin.

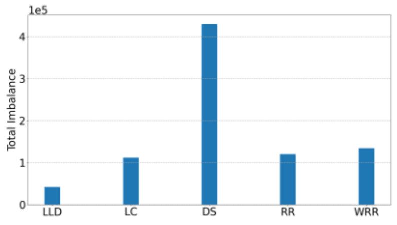


Fig. 11 Total imbalance for medium workload

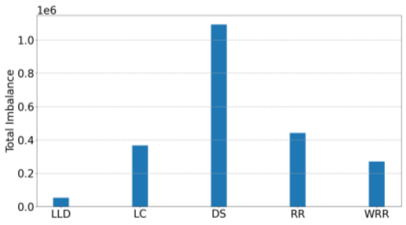


Fig. 12 Total imbalance for Heavy workload

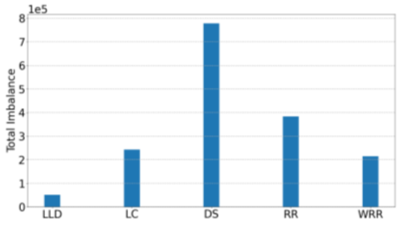


Fig. 13 Total imbalance for mixed workload

9.2.3. Finishing Time

Fig. 14, *Fig. 15*, and *Fig. 16* depict the total execution time of all the submitted applications in the three scenarios. It is clear that our proposed approach completes the execution of all the submitted workload faster than the other approaches.

- In *Fig.14*: For a medium workload, "LLD" is 2.5 times faster than Least Connections, 2.8 times faster than Device Suitability, 2.8 times faster than Round Robin, and 3.3 times faster than Weighted Round Robin.
- In *Fig.15*: For a Heavy workload, "LLD" is 2.3 times faster than Least Connections, 5.4 times faster than Device Suitability, 3.1 times faster than Round Robin, and 2.1 times faster than Weighted Round Robin.
- In *Fig.16*: For a mixed workload, "LLD" is 3.2 times faster than Least Connections, 6.3 times faster than Device Suitability, 3.7 times faster than Round Robin, and 3 times faster than Weighted Round Robin.

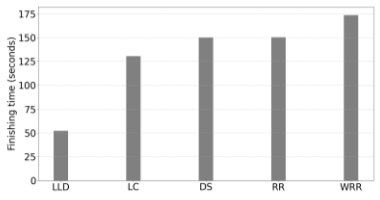


Fig. 14 Finishing time for medium workload

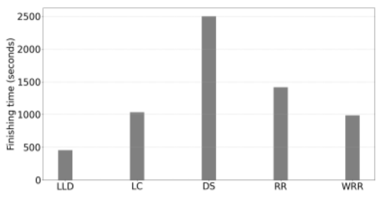


Fig. 15 Finishing time for heavy workload

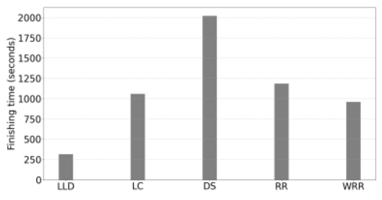


Fig. 16 Finishing time for mixed workload

9.2.4. Device Total Execution Time

Fig. 17, *Fig. 18*, and *Fig. 19* present the load balancing analysis of the scheduling approaches. These figures illustrate the total time during which each device was executing computational tasks. Our proposed approach attains superior load balancing performance by achieving an equitable distribution of work across the devices. This efficient load distribution ensures that each device is optimally used, leading to improved system performance and reduced waste of resources.

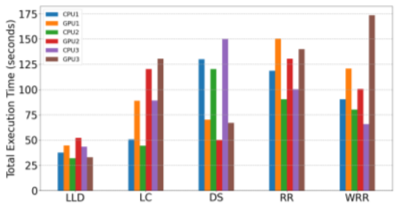


Fig. 17 Load balance analysis for medium workload

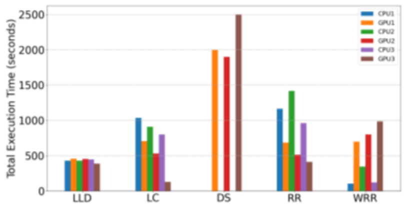


Fig. 18 Load balance analysis for heavy workload

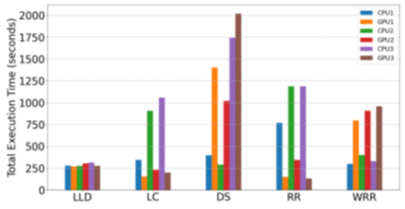


Fig. 19 Load balance analysis for mixed workload

9.2.5. Device Idle Time

Fig. 20, *Fig.* 21, and *Fig.* 22 illustrate the idle time of each device, representing the duration when a device remains operational but does not perform any computing tasks. This idle time signifies wasted energy during the experiments. Our proposed approach demonstrates superior performance in reducing idle time, leading to significant energy savings and enhanced overall energy efficiency in the system.

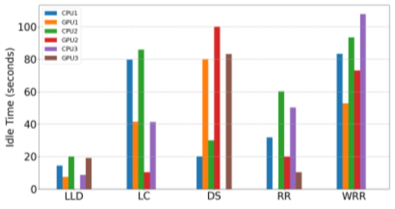


Fig. 20 Devices' idle time for medium workload

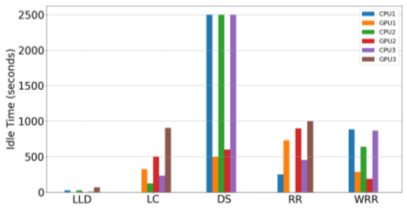


Fig. 21 Devices' idle time for heavy workload

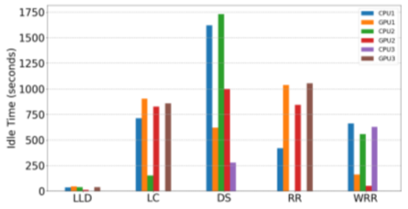


Fig. 22 Devices' idle time for mixed workload

9.2.6. *Energy consumption*

The power consumption per second when the devices are idle was measured using the tools perf [31] for Intel processors and nvidia-smi [32] for Nvidia GPUs. It should be noted that the idle power of a device can be affected by a variety of factors, including the CPU and GPU power management settings, background processes, connected peripherals, system configuration, BIOS/UEFI settings, and operating system settings. As a result, the idle power values vary from one system to another.

Table 21 summarizes the energy consumption of the system's devices in both their idle and maximum power consumption states.

Table 21. Device's energy consumption

Device	Model	Active Energy (Watts)	Idle Energy (Watts)
CPU1	Xeon(R) CPU E3-1225	95 [27]	29
GPU1	Quadro P400	30 [28]	11
CPU2	Xeon(R) CPU E3-1225	95 [27]	29
GPU2	Quadro P400	30 [28]	11
CPU3	i7-6700	65 [29]	20
GPU3	Quadro K620	45 [30]	15

Fig. 23, *Fig.* 24, and *Fig.* 25 illustrate the power consumption of all system devices throughout the entire experiment.

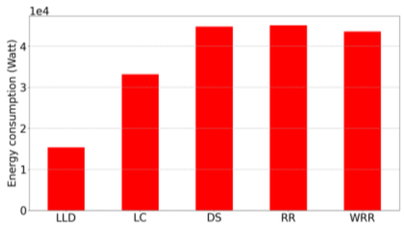


Fig. 23 Devices' energy consumption for medium workload

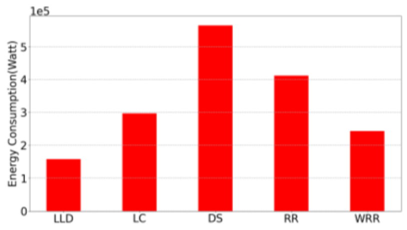


Fig. 24 Devices' energy consumption for heavy workload

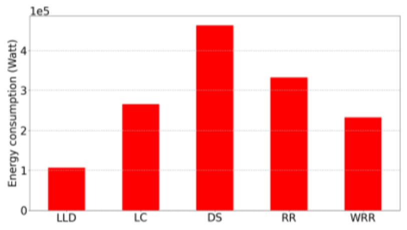


Fig. 25 Devices' energy consumption for mixed workload

- According to *Fig.*23, for a medium workload, "LLD" reduces the energy consumed by 53 % less than Least Connections, 65% less than Device Suitability, 65% less than Round Robin, and 64% less than Weighted Round Robin.
- According to *Fig.* 24, for a heavy workload, "LLD" reduces the energy consumed by 46% less than Least Connections, 71% less than Device Suitability, 61% less than Round Robin, and 35% less than Weighted Round Robin.
- According to *Fig.* 25, for a mixed workload, "LLD" reduces the energy consumed by 59% less than Least Connections, 75% less than Device Suitability, 67% less than Round Robin, and 54% less than Weighted Round Robin.

Table 22, *Table 23*, and *Table 24* provide the numerical values of the experimental results for the 3 different experiment scenarios**.**

Table 22. Experimental results for medium workload

Approach	Total imbalance	Finishing time (Seconds)	Power consumption (Watt)
LLD	42215.18	52.14	15383.55
LC	111715.32	130.52	33206.20
DS	430089.46	150.10	44836.78
RR	120615.92	150.52	45160.01
WRR	134131.30	173.63	43600.99

Table 23. Experimental results for heavy workload

Approach	Total imbalance	Finishing time (Seconds)	Power consumption (Watt)
LLD	53087.24	456.746	158321.35
LC	367859.51	1059.22	297429.65
DS	1093745.81	2500.78	564789.15
RR	441683.91	1416.64	411773.46
WRR	271180.46	988.45	243817.52

Table 24. Experimental results for mixed workload

Approach	Total imbalance	Finishing time (Seconds)	Power consumption (Watt)
LLD	51279.77	316.16	106645.58
LC	243116.34	1034.97	265577.92
DS	878783.06	2022.34	436728.39
RR	383504.56	1188.06	332589.68
WRR	214148.86	960.66	233042.34

9.3. Results Discussion

The experimental results demonstrate that our proposed scheduling scheme outperforms all other scheduling schemes in terms of load balance, execution time, and energy efficiency in all the different experiment cases. The results proved that balancing the load leads to faster execution and less energy consumption.

In this section, we will discuss the performance of every scheduling scheme across the various scenarios.

9.3.1. Least Connections (LC)

The Least Connections scheduling approach assigns applications to the device with the fewest tasks in its queue. This approach favors faster devices.

When the workload has moderate computational requirements, the CPU is often faster than the GPU due to the overhead of copying data from central memory to GPU memory. The speedup gained from GPU execution is not significant enough to compensate for this overhead. Considering only the number of applications in the devices' queues leads to overloading the GPUs, as they execute slower even when processing a smaller number of applications compared to the CPUs (*Fig. 17*).

In contrast, in scenarios with heavy workloads, the applications execute much faster on the GPUs than on the CPUs. The time taken by the CPU to execute one application can be equivalent to the time taken by the GPU to execute multiple applications. In this case, the Least Connections method overloads the CPUs, as they take too much time to handle heavy applications (*Fig. 18*).

When the workload consists of both heavy and moderate applications, the Least Connections approach overloads the CPUs. Assigning heavy applications to the slower CPUs results in a much larger cost compared to assigning medium applications to the GPUs (*Fig. 19*).

The imbalance in the three scenarios is caused by the fact that the Least Connections scheduling approach disregards the characteristics of the applications and the computing devices. It limits the definition of the load to the number of applications when assigning them to devices which can be inaccurate as a measure of the system load. A device may be overloaded even with a small number of applications if those applications are computationally intensive or not well-suited for the device. Conversely, a device with many applications may not be overloaded if those applications are light and do not strain its resources.

As a result, the Least Connections scheduler is best suited for homogeneous systems with workloads that are primarily composed of similar applications with similar computational intensity.

9.3.2. Device Suitability (DS)

The Device Suitability scheduler assigns applications to the device with the highest processing power, regardless of the other applications that are pending on the device. This approach aims to reduce the execution time of each application individually. However, this approach can lead to an imbalance in the system.

For instance, when the workload is moderate, the CPUs are better suited to execute the applications in most cases. The Device Suitability scheduler will overload the CPUs while the GPUs remain underutilized (*Fig. 17*). In contrast when the workload is heavy, the CPUs are completely ignored, resulting in GPU overloading (*Fig. 18*).
In the case of a mixed workload, the Device Suitability approach overloads the most powerful CPU with applications of moderate computing requirements and the most powerful GPU with intensive applications (*Fig. 19*).

Disregarding the load of the devices while assigning applications in the Device suitability resulted in longer queues for the powerful computing devices and imbalanced distribution of the workload.

The Device Suitability scheduler can produce better performance when the workload is evenly balanced, and all devices in the system handle similar loads of applications with proportional computational demands

9.3.3. Round Robin (RR)

The Round Robin scheduler adopts a straightforward strategy in which applications are cyclically assigned to system devices, without considering the devices' current load state or their computational capabilities. Consequently, this approach may lead to less efficient allocations, with computationally intensive applications assigned to slower devices and less demanding applications to more powerful devices. As a result, workload distribution can become imbalanced, impacting the overall performance of the system (*Fig. 17, Fig. 18, and Fig. 19*).

The imbalance in workload distribution becomes more apparent when the submitted workload includes intensive applications when the workload is heavy or mixed (*Fig.18 and Fig. 19*). In such cases, the impact of assigning a heavy application to an unsuitable device is way more costly.

In summary, the Round Robin approach is most suitable for homogeneous systems having computing devices with comparable processing power, and workloads consisting of applications with similar computing intensities.

9.3.4. Weighted Round Robin (WRR)

The Weighted Round Robin (WRR) scheduler is an extension of the Round Robin approach, which introduces device weights based on their computing power. In this approach, tasks are assigned proportionally to the device weights.

In scenarios where the workload is heavy or comprises computationally intensive applications, the WRR scheduler provides better performances (*Fig. 18 and Fig. 19*) by allocating a higher number of applications to more powerful devices. This results in a more balanced workload distribution across the computing resources, as the powerful devices can handle the heavier computational demands better.

However, the performance of the WRR scheduler may be less optimal when the workload is moderate and consists of applications with moderate computing requirements. In such cases, the scheduler can overload the more powerful devices with applications that are not suitable for their processing capabilities, leading to suboptimal system performance.

For better results, the Weighted Round Robin approach is best suited for deployment in homogeneous systems where devices have similar architectures but different computing powers, such as systems comprising only GPUs. In these scenarios, the device weights can be assigned based on their computational capabilities, ensuring that applications are efficiently distributed among the devices according to their processing requirements.

9.3.5. Least Loaded (LLD)

The experimental results demonstrate the impracticality of the cited approaches in a heterogenous system, where the computing devices have different computing power and architectures and the workload is dynamic and imbalanced. This impracticality is because each scheduling method ignored one or multiple factors that are crucial in the performance of heterogeneous systems, including the device architectures, the applications' nature and the type of computations it performs (sequential or parallel) and the load definition of the devices. The imbalance resulted from these schedulers has led to longer execution time (*Fig.14, Fig.15 and Fig. 16*) as the overloaded devices finish the execution late resulting in more idle time for the underloaded device (*Fig. 20, Fig. 21 and Fig. 22*). Longer execution and idle times resulted in more energy consumption (*Fig. 23, Fig. 24 and Fig. 25*).

In this study, we address the limitations observed in existing schedulers for heterogeneous systems. We propose a new scheduler that assigns each submitted application to an underloaded device. To achieve this, we introduce a comprehensive definition of the load that captures all relevant aspects, providing a precise representation of the device's actual workload. These factors include the characteristics of the devices as well as the computational requirements of the applications.

Our scheduler continuously monitors the system in real-time and updates the load of each device accordingly. It adopts a strategy of favoring underloaded devices for application assignment. This preference ensures a balanced workload distribution by reducing the load differences between devices (*Fig. 11, Fig. 12, and Fig. 13*). As demonstrated in *Fig. 8, Fig. 9, and Fig. 10,* the imbalance curves consistently converge towards zero throughout the experiment, validating the effectiveness of our load balancing method.

The balanced load distribution achieved by our approach offers several significant advantages. Firstly, it results in reduced energy consumption (*Fig. 23, Fig. 24, and Fig. 25*), as devices are efficiently utilized, and idle time is minimized (*Fig. 20, Fig. 21, and Fig. 22*). Additionally, the execution time of the workload is significantly decreased, as applications are assigned to devices based on their current load status (*Fig. 14, Fig. 15 and Fig. 16*).

In summary, our proposed approach addresses the limitations of existing schedulers by providing a sophisticated load balancing mechanism that optimizes the utilization of resources in heterogeneous systems. By considering multiple factors in the load measurement and continuously adapting to the system's dynamics, our approach achieves consistent load balance and improved overall system performance.

9.3.6. Overhead

The conducted experiments reveal that the overhead resulting from features extraction and execution time prediction is negligible. This can be attributed to the integration of features extraction into the application compilation process, which optimizes the efficiency of the overall process. Furthermore, the execution time prediction models are directly applied, leveraging their pre-existing training, leading to reduced computational overhead.

It is worth mentioning that the execution time models undergo a one-time training process. Subsequently, these models remain operational for the entire duration of their usage without requiring repeated training. This approach ensures consistent and efficient execution time predictions for all subsequent applications without the need for repeated training, thereby optimizing the overall performance of the system.

10. Conclusion

Achieving equitable distribution of the computing load across system devices is essential for maintaining system balance and optimizing performance while minimizing power consumption. This need for load balancing becomes particularly evident when scheduling applications in real-time heterogeneous systems. These systems pose unique challenges as applications are time-sensitive and have immediate computing resource demands, while prior information about these applications is absent.

In this article, we proposed a real-time CPU-GPU heterogeneous cluster resource manager that aims to address load balancing challenges. The core concept of our approach involves incorporating the predicted execution time into the definition of the device's load. By doing so, our scheduler can efficiently detect load imbalances, identify underloaded devices, and subsequently assign applications to these underloaded devices to achieve load balancing.

To measure the system's performance in terms of load balancing, we introduced a novel metric. This metric is calculated based on our definition of the system loads and provides a comprehensive measure of the system's load distribution. By using this metric, we quantitatively evaluate the effectiveness of our load balancing approach and compare it with alternative methods.

When compared to alternative popular scheduling schemes from various domains, including Round Robin, Weighted Round Robin, Least Connections, and Device Suitability scheduler, our proposed approach exhibits notable improvements in both system balance and execution time. It surpasses other scheduling schemes by achieving the least load imbalance and execution time, resulting in reduced energy consumption.

However, the "LLD" approach presents a platform dependency, necessitating retraining when deployed on alternative systems. Moreover, its performance is directly affected by the accuracy and precision of the employed machine learning models. Furthermore, the inclusion of applications' source code raises confidentiality concerns, as it may expose proprietary or sensitive information.

To address the limitation of the "LLD" approach, the following enhancements can be implemented:

1. Generalizing prediction models by adopting a unified prediction model for all computing devices. This generalization will enable the "LLD" approach to handle various hardware configurations effectively, resulting in improved adaptability across diverse platforms.
2. Enhancing Machine Learning precision by incorporating a broader range of application types in the training dataset. This allows the "LLD" approach to better handle different computational workloads, leading to improved prediction accuracy of execution times.
3. Addressing the requirement of the source code by providing users with a features extractor script. This script extracts the necessary features during the compilation process of the users' applications. As a result, users can submit their compiled applications along with the extracted features, eliminating the need for source code. This approach maintains data security and confidentiality while enabling the "LLD" scheduler to access the required information for its effective functioning.

11. References

[1] Borja P., Esteban S., José L. B., Ramón B., "Energy efficiency of load balancing for data-parallel applications in heterogeneous systems",

The Journal of Supercomputing, Vol. 73, pp. 330–342, 2017, https://doi.org/10.1007/s11227-016-1864-y

[2] Vella F., Neri I., Gervasi O., Tasso S., “A simulation framework for scheduling performance evaluation on CPU-GPU Heterogeneous System”, International Conference on Computational Science and Its Applications, Springer, Berlin, Heidelberg, 2012, https://doi.org/10.1007/978-3-642-31128-4_34

[3] Khalid Y.N., Aleem M., Usman A., Muhammad A. I., Islam M. A., Iqbal M. A., “Troodon A machine-learning based load-balancing application scheduler for CPU–GPU system”, Journal of Parallel and Distributed Computing, Vol. 132, pp. 79-94, 2019, https://doi.org/10.1016/j.jpdc.2019.05.015

[4] Mahmoudi Sidi., Manneback P., Augonnet C., Thibault S., “Traitements d'images sur architectures parallèles et hétérogènes“, Techniques et sciences informatiques (Computer science and technology), Vol. 31, pp. 1183-1203, 2012 https://doi.org/10.3166/tsi.31.1183-1203

[5] Olivier V., Pangfeng L., Jan-Jan W., “A collaborative CPU–GPU approach for principal component analysis on mobile heterogeneous platforms”, Journal of Parallel and Distributed Computing, Vol. 120, pp. 44-61, 2018, https://doi.org/10.1016/j.jpdc.2018.05.006

[6] Harichane I., Makhlouf SA., Belalem G., “KubeSC-RTP: Smart scheduler for Kubernetes platform on CPU-GPU heterogeneous systems”. Concurrency and Computation Practice and Experience e7108, 2022 https://doi.org/10.1002/cpe.7108

[7] Grauer-Gray S., Xu L., Searles R., Ayalasomayajula S., Cavazos J., "Auto-tuning a high-level language targeted to GPU codes, Innovative Parallel Computing, pp. 1-10, 2012, https://doi.org/10.1109/InPar.2012.6339595

[8] Moez A., (2020). PyCaret: An open source, low-code machine learning library in Python. https://www.pycaret.org/

[9] Borja P., Stafford E., Bosque J.L., Beivide R., “Sigmoid: an auto-tuned load balancing algorithm for heterogeneous systems”, Journal of Parallel and Distributed Computing, Vol. 157, pp. 30-42, 2021, https://doi.org/10.1016/j.jpdc.2021.06.003

[10] Khalid Y.N., Aleem M., Prodan R., Iqbal M.A, Islam M. A., “E-OSched: a load balancing scheduler for heterogeneous multicores”. Journal of Supercomputing, Vol. 74, pp. 5399–5431, 2018, https://doi.org/10.1007/s11227-018-2435-1

[11] Usman A., Jerry C.W. L., Gautam S., Aleem M., "A load balance multi-scheduling model for OpenCL kernel tasks in an integrated cluster", Soft Computing - A Fusion of Foundations, Methodologies and Applications, Vol. 25, pp. 407–420, 2021 https://doi.org/10.1007/s00500-020-05152-8

[12] Judit P., Rosa M.B., Eduard A., Jesús L., "SSMART: smart scheduling of multi-architecture tasks on heterogeneous systems", WACCPD : Proceedings of the Second Workshop on Accelerator Programming using Directives, pp. 1-11, 2015, https://doi.org/10.1145/2832105.2832109

[13] Yong M.T. and Rassul A., "Comparison of Load Balancing Strategies on Cluster-based Web Servers", SIMULATION: Transactions of The Society for Modeling and Simulation International, Vol. 77, Issue 5-6, pp. 185–195, 2001, https://doi.org/10.1177/003754970107700504

[14] Taha A. R., Fatima D., Ghalem B., Sidi Ahmed M., "HBalancer: A machine learning based load balancer in real time CPU-GPU heterogeneous systems", International Conference on Innovation and Intelligence for Informatics, Computing, and Technologies (3ICT), Bahrain, pp. 674-679, 2022, https://doi.org/10.1109/3ICT56508.2022.9990623

[15] Khronos Group, OpenCL Specification, version 2.2. (2019), https://registry.khronos.org/OpenCL/

[16] Tarek H., Sadam A., Omar B., "A machine Learning-Based approach to estimate the CPU-Burst time for processes in the computational grids," in International Conference on Artificial Intelligence, Modelling and Simulation (AIMS), 2015, https://doi.org/10.1109/AIMS.2015.11

[17] Adrian S.,"LLVM for Grad Students", 2015, https://www.cs.cornell.edu/~asampson/blog/llvm.html.

[18] Pycaret Documentation, https://pycaret.gitbook.io/docs/get-started/functions

[19] Bestavros A., “WWW traffic reduction and load balancing through server-based caching”, IEEE Concurrency, vol. 5, no. 1, pp. 56-67,1997 https://doi.org/10.1109/ACCESS.2021.3065170

[20] Lung-Hsuan H., Chih-Hung W., Chiung-Hui T., Hsiang-Cheh H., “Migration-Based Load Balance of Virtual Machine Servers in Cloud Computing by Load Prediction Using Genetic-Based Methods”, IEEE Access, vol. 9, pp. 49760-49773, 2021, https://doi.org/10.1007/978-3-642-19861-8_16

[21] Wen Y., Wang Z., O’Boyle M. F. P, “Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms”. International Conference on High Performance Computing (HiPC), 2014, https://doi.org/10.1109/hipc.2014.7116910

[22] Cybenko G., “Dynamic load balancing for distributed memory multiprocessors”, Journal of Parallel and Distributed Computing, Vol. 7, No. 2, pp. 279–301, 1989, https://doi.org/10.1016/0743-7315(89)90021-x

[23] Hong J. C., Dong O. S., Seung G. K., Jong M. K., Hsien-Hsin L., Cheol H. K., “An efficient scheduling scheme using estimated execution time for heterogeneous computing systems”, The Journal of Supercomputing, Vol 65, pp. 886–902, 2013, https://doi.org/10.1007/s11227-013-0870-6

[24] Audace M.; Saadi B.; Lamia C. F., “A Priority-Weighted Round Robin scheduling strategy for a WBAN based healthcare monitoring system”, 13th IEEE Annual Consumer Communications & Networking Conference (CCNC), pp. 224-229, 2016, https://doi.org/10.1109/CCNC.2016.7444760

[25] Weikun W., Giuliano C.,” Evaluating Weighted Round Robin Load Balancing for Cloud Web Services”, 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, pp. 393-400, 2014, https://doi.org/10.1109/SYNASC.2014.59

[26] Bhavya A., Gaurav S., Harleen K., Raul V., Victor C., “Experimental Setup for Investigating the Efficient Load Balancing Algorithms on Virtual Cloud”, Sensors, no. 24: 7342, 2020, https://doi.org/10.3390/s20247342

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

[28] Nvidia Corporation, 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

[29] Intel Corporation, https://ark.intel.com/content/www/fr/fr/ark/products/88196/intel-core-i76700-processor-8m-cache-up-to-4-00-ghz.html

[30] Nvidia Corporation, https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/documents/75509_DS_NV_Quadro_K620_US_NV_HR.pdf

[31] Ubuntu manuals, https://www.man7.org/linux/man-pages/man1/perf.1.html

[32] Nvidia Corporation, https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf

[33] Konrad M., Diana G., “Automatic Mapping for OpenCL-Programs on CPU/GPU Heterogeneous Platforms”, International Conference on Computational Science ICCS, pp. 301–314, 2018 , Springer, Cham, https://doi.org/10.1007/978-3-319-93701-4_23

[34] Lee J., Samadi M., Park Y., Mahlke SA., “Transparent CPU-GPU collaboration for data-parallel kernels on heterogeneous systems”, 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK, 2013, pp. 245-255, 2013, https://doi.org/10.1109/PACT.2013.6618821

[35] Chen R. C., Dewi C., Huang S., Caraka R., “Selecting critical features for data classification based on machine learning methods”. Journal Of Big Data, 2020, https://doi.org/10.1186/s40537-020-00327-4

[36] Chen X., Jeong J. C., “Enhanced recursive feature elimination”, 2007, Sixth International Conference on Machine Learning and Applications ICMLA , 2007, pp. 429-435, https://doi.org/10.1109/ICMLA.2007.35

[37] Chicco D., Warrens M., Jurman G., “The coefficient of determination R-squared is more informative than SMAPE, MAE, MAPE, MSE and RMSE in regression analysis evaluation”, PeerJ Computer Science 7: e623, 2021, https://doi.org/10.7717/peerj-cs.623