Received: 21 December 2019 | Revised: 18 August 2020 | Accepted: 19 August 2020

DOI: 10.1002/spe.2898

RESEARCH ARTICLE

WILEY

KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters

Ghofrane El Haj Ahmed | **Felipe Gil-Castiñeira** | **Enrique Costa-Montenegro**

atlanTTic Research Center for Telecommunication Technologies, Universidade de Vigo, Vigo, Spain

Correspondence
Felipe Gil-Castiñeira, atlanTTic Research Center for Telecommunication Technologies, Universidade de Vigo, Vigo, Spain.
Email: xil@gti.uvigo.es

Summary

Container platforms are increasingly being used to deploy cloud-based services. Nevertheless, many cloud services are also demanding graphics processing units (GPUs) to accelerate different applications that make use of their parallel architecture, such as deep learning or just video processing. Thus, different container technologies, such as Docker and Kubernetes, are implementing GPU support. Some effort is being devoted to design algorithms to schedule applications into heterogeneous computing systems that use CPUs and GPUs together. This article is part of this effort, and we describe how to build a dynamic scheduling platform for Kubernetes that is able to manage the deployment of Docker containers in a heterogeneous cluster, which we call KubCG. This platform implements a new scheduler that optimizes the deployment of new containers by taking into account the Kubernetes Pod timeline and the historical information about the execution of the containers. We have performed different tests to validate this new algorithm, and KubCG was able to reduce the time to complete different tasks down to a 64% of the original time in our different experiments.

KEYWORDS

cloud computing, CPU, Docker, GPU, Kubernetes

1 | INTRODUCTION

The cloud computing paradigm allows reducing infrastructure costs by offering a pool of devices as a service to different customers. Large datacenters are dimensioned to typically satisfy the demands of their users, but nowadays, the multi-access edge computing (MEC) paradigm is pushing the creation of smaller and distributed infrastructures near the final users, in order to provide low latency services, which should be small in order to make their installation affordable.[1] MEC applications usually provide services that cannot be implemented in the user device, because of the large amount of resources required, that should return results with very low latency. For example, deep learning algorithms that classify video in real time and that require a powerful graphics processing unit (GPU) to be executed.[2]

Thus, it is necessary to optimize the usage of resources with new scheduling algorithms that take into account the particular resources required by the application (e.g., applications may require a GPU, or just use it to complete their tasks sooner).

Virtualization is one of the key technologies that are making possible to implement an efficient MEC architecture. By using virtualization techniques, the cloud platforms can be much easily shared between different users or different services, maximizing the usage of the hardware and making it possible to deploy a large amount of MEC sites.

Current virtualization techniques that are commonly used are hypervisors and containers.[3] Hypervisor-based systems create virtual machines that run a guest operating system. The hypervisor presents the guest operating systems with a virtual operating platform, so it is even possible to run different operating systems in parallel. Container-based systems work at the operating system level. The kernel of the operating system allows multiple user space instances (the containers), which are isolated from other instances or from the systems resources. Container-based virtualization imposes less overhead than hypervisor-based systems, making them gain a lot of popularity in the last years.

Containers are a relatively low-level construct that provides applications a view of the operating system running on the physical device. Nevertheless, the developers of the container environments usually provide additional tools to, for example, configure, build, or manage the different containers. Furthermore, containers can be deployed on clusters. For example, Kubernetes[1] and Docker Swarm[2]decouple containers from the details of the systems where they run, and orchestrate and schedule containers to use those resources.[3]

Having into account that there are applications that can take advantage of hardware accelerators (such as GPUs), container orchestrators should be able to share GPUs among them. Nevertheless, virtualizing GPUs has been considered more difficult than virtualizing other devices. For example, the distribution of the same GPU among several virtualized systems presented several challenges, such as the fact that GPUs do not provide information regarding how long a request occupies the GPU, or that GPUs are usually nonpreemptive (a virtual system cannot be preempted by software until it finishes).[4] Furthermore, GPU virtualization for container-based systems is still in its early stage. For example, Docker[3](one of the most popular container runtimes) did not include native support for GPUs until version 19.03, which integrated the tools provided by NVIDIA to make it possible for applications running in a container to use the GPUs of this manufacturer. Nevertheless, fractional sharing is not supported, and the usage of the GPU is limited to a single application at a time. This is a fundamental problem shared by the different containerization technologies, operating systems and GPU devices.

In addition, many advanced MEC applications (such as video processing or machine learning) will require GPU acceleration, and many MEC architectures are providing support for containers, thus making it necessary to find mechanisms to efficiently share GPUs among containers.

A simple but effective approach to optimize the use of GPU nodes in a cluster consists in implementing a scheduler that distributes applications (or Pods, in Kubernetes) to the most appropriate node according to the requirements of the Pod and the available resources. Consequently, we should consider efficient scheduling methods for heterogeneous cloud systems (clusters that include nodes with different hardware). The Kubernetes default scheduler will assign a Pod to a Node, but this is a static procedure that is performed only when the Pod is created by a user or a controller, resulting in a nonoptimal system performance. A dynamic scheduling scheme that also considers the features of the Pod and if a node includes a GPU, can improve the performance and the global efficiency of the system. This is especially important for private clouds with a limited number of resources that have to be carefully assigned.

Our article describes how Docker and Kubernetes can be modified and configured to deploy applications and optimize the usage of hardware accelerated nodes. To achieve this objective, we have designed a new scheduler that uses the information about the previous executions of a Pod to estimate the needs to complete the new task and decide where it should be deployed.

The main contributions of this article are as follows:

- The creation of a model for the lifecycle of a Pod, which will be used by the scheduler to plan the execution of the different Pods.
- The design and development of an entity ("switch") that automatically selects an image for a CPU or GPU node according to the result of a decision.
- The creation of a database that collects statistics about the usage of resources by a particular Pod. This information is used later by the scheduler to decide the best option to deploy the Pod in future executions.
- A new algorithm that analyses a queue of Pods and assigns them to the different nodes of an heterogeneous cluster, optimizing the overall computational throughput. This algorithm estimates the time to complete the execution of a Pod using the different nodes and resources and selects the nodes that are compatible with the Pod and that will minimize the time to complete the tasks. The algorithm also analyses if it is worth waiting for nodes that are already executing another Pod.

[1]https://kubernetes.io
[2]https://docs.docker.com/engine/swarm/
[3]https://www.docker.com

- The support of older GPUs, as some old GPUs cannot be used by the default scheduler.
- The development of a benchmarking environment to validate the improvement provided by our algorithm compared with the default Kubernetes scheduler.

This article is organized as follows: Section 2 introduces background information regarding the containerization ecosystem and scheduling in Kubernetes. Section 3 reviews related research work regarding scheduling in heterogeneous cluster systems and in container ecosystem. Section 4 models the execution of a Pod in Kubernetes. This model is later used by the KubCG scheduler described in Section 5. Section 6 presents the environment (hardware and software) used to validate our proposal, describes the experiments completed, the results obtained and a comparison between the performance of the KubGC scheduler and the default one used in Kubernetes. Finally, Section 7 concludes the article and introduces some future research lines.

2 | CONTAINERIZATION ECOSYSTEM AND SCHEDULING IN KUBERNETES

Container-based virtualization is an approach to isolate guest applications in which the virtualization layer runs as a layer within the operating system. The kernel runs on the hardware node with several isolated guests on top of it. Thus, this is a lightweight approach in comparison with the hypervisor-based systems. The principal idea is to deploy instances that share a single host operating system and relevant binaries, libraries or hardware drivers.

Docker is the most popular open source container engine, which has become widely used in the last years. Docker Engine, Docker Containers, Docker Images, Docker Client, and Docker Daemon are the principal components of Docker. The Docker Engine allows the creation and execution of the Docker Containers, where the container is a live instance of a Docker Image. Containers are controlled and managed by the Docker Daemon. Users can communicate with the Docker Engine by using the Docker Client.

Nvidia–Docker [4]is a Docker extension that is designed for the containerization of CUDA Images and their execution in Nvidia GPUs. CUDA is an API created by Nvidia for using GPUs for general purpose processing.[5]

Containerization has proved that it can replace the virtual machines efficiently in many scenarios. Some articles compare hypervisor-based virtual machines and containers. For instance, Felter et al.[6] compared the performance of traditional virtual machines deployments (using KVM as a representative hypervisor) with Docker as a container manager. In addition, Kämäräinen et al.[7] compared the performance of virtual machines and containers in cloud gaming systems.

Kubernetes is an open source platform that automates the use of containers. It can eliminate many manual processes associated with deploying, scaling, and management containerized applications. The main components of Kubernetes are:

- Master server: a server that controls Kubernetes worker nodes. The master server manages and schedules the different resources existing in the cluster such as storage, CPUs, GPUs, and so forth.
- Worker nodes: machines that perform the assigned tasks. Each worker node can run one or multiple Pods.
- Pod: is a unit of deployment that encapsulates a single instance of an application. It is the basic working unit in Kubernetes. Each Pod can encapsulate a single container (the most common case) or a set of containers. It is scheduled to run on one node, with all containers inside the Pod being deployed on the same machine and sharing the same IP address, host name, and other resources. Furthermore, different deployment rules can be set for Pods using "Workloads." For example, "Jobs" are used to implement applications that may launch one or more Pods to complete a particular task and terminate. Nevertheless, Kubernetes is also used to implement applications that are expected to run indefinitely and to provide a service to external user or clients. In that scenario a "Service" abstraction provides the connectivity with the external world and manages a set of Pods which implement the functionality.

The Kubernetes scheduler is the part that is responsible to schedule Pods onto nodes. Basically, when a new Pod is created, or when it is restarted, the scheduler selects a node from the list of the nodes available in the cluster and assigns the Pod to that node.

[4]https://github.com/NVIDIA/nvidia-docker

The default Kubernetes scheduler processes two types of policies to select the node to run the Pod:

- Fit predicate: it is a function that can choose the node depending on some factor like the node selector (the user can specify some condition that the scheduler should respect in order to select the node for the new Pod) and that there are enough resources that allow the execution of the Pod. For instance, the user can indicate the "node selector" to choose the nodes that have some specific features like the CPU model.
- Priority function: after the filtration process, the scheduler can find that multiple machines are "fit" for the new Pod. However, the Pod can only be scheduled onto one machine. This function ranks the machines by using some factors, for example: it prioritizes the machines whose already-running Pods consume the least resources.

3 | RELATED WORK

In this work, we describe how to modify Kubernetes to maximize the use of hardware accelerators, such as GPUs, to optimize the execution of applications that may take advantage of these devices, such as machine learning or video processing applications. First of all, it is necessary to allow the virtualized application to use and share the hardware device.

GPU virtualization is gaining considerable attention in the research community due to the increase usage of these devices in different areas. The majority of the researchers have focused on providing access to GPU accelerators within virtual machines (hypervisor-based virtualization). Gupta et al.[8] presented GViM, a software architecture that allows CUDA applications to run within VMs under the Xen virtual machine monitor. In this application, the management OS that called all dom0 controls the GPU in Xen parlance. The actual GPU device driver therefore executes in dom0, while applications run on the guest OS or VM. By using a split-driver model, the CUDA run-time API function calls from the application are captured by an interposer library within the VM, followed by transfer of the function parameters to dom0 for actual interaction with the GPU. Shi et al.[9] presented a first CUDA-oriented GPU virtualization solution, called vCUDA. They additionally provide suspend and resume facility by maintaining a record of the GPU state within the virtualization infrastructure in the guest OS as well as the management OS domain. Duato et al.[10] designed rCUDA, a software framework that can enable sharing GPUs remotely in HPC clusters. This framework allows a reduction in the number of accelerators installed in the cluster to minimize the power consumption. Diab et al.[11] presented a framework which called "gloud," which enables on-demand use of GPUs in a cloud computing environment. This framework allows the registration and the execution of GPU kernels from multiple applications concurrently on distributed GPUs. Zhoa et al.[12] built a dynamic GPU resources allocation and management framework based on multiple load features of the GPU in the cloud using the virtual machines. This framework allows an automatic detection of the nodes in the cloud and estimate the computing capacity of GPU dynamically to reach the purpose of the rational allocation of the GPU in the cloud environments.

As introduced in Section 1, Nvidia has distributed Nvidia–Docker to build and run GPU accelerated containers. Nevertheless, sharing GPUs among multiple containers works like sharing GPUs among multiple processes outside of containers. Traditionally, only one CUDA context could be run simultaneously (they have to be executed sequentially). Although Nvidia created the multiprocess service, an alternative implementation of the CUDA interface that transparently enables cooperative multiprocess CUDA applications,[13] this feature is still not supported by Nvidia–Docker [5]. Therefore, it is necessary to implement a mechanism to optimize the distribution of tasks and Pods among the available GPUs.

The design of efficient schedulers to improve the performance of heterogeneous computing and cloud systems is also a topic that has been addressed by researchers. Jiménez et al.[14] proposed a new architecture of predictive user-level scheduler that used the past performance history to select between using nodes with just a CPU or with a CPU and a GPU. Choi et al.[15] also designed a dynamic scheduler that selects between a CPU and GPU in a heterogeneous platform, using the execution history. The selection is based on the estimated execution time and remaining time. Gregg et al.[16] described other dynamic scheduling algorithm that schedules OPENCL applications by using a historical database of runtimes values and the selection between the CPU and GPU depends on the data size of the input data. Furthermore, Wen et al.[17] introduced a scheme for OPENCL scheduler that can schedule multiple programs on a heterogeneous computing system using a prediction technique. Schulga et al.[18] based the design of the scheduler in a machine-learning algorithm.

[5]https://github.com/NVIDIA/nvidia-docker/wiki#do-you-support-cuda-multi-process-service-aka-mps

Rafique et al.[19] built a dynamic scheduler model which collects the information about the previous performance and decides which of the available resources to use, assuming optimized implementations of the same application targeted to different available resources and can be dynamically sent to any one of them as needed. Trejo-Sánchez et al.[20] presented a scheduler of clusters based on a multiagent architecture. Other researchers have identified several research challenges that are addressed in this article. A recent publication by Rodriguez et al.[21] identifies the need for cloud-aware placement algorithms to consider the heterogeneities of the underlying resources (including parameters such as resource types and sizes). Joseph et al.[22] state that resource request profiling and estimation is a research challenge that will aid the allocation and scheduling processes in the microservice systems.

Regarding containerization, Civolani et al.[23] proposed a solution to optimize start-up container deployment times. Medel et al.[24] analyzed performance of the container orchestration using the Kubernetes system and developed a reference net-based model of resource management within this system using Petri nets. In addition, on the related subject of the container scheduling in cloud environment, Victor et al.[25] studied how the default scheduler can cause the performance degradation and they designed an approach called client-side scheduling to solve this issue. Cérin et al.[26] proposed an architecture for a new scheduler for Docker Swarm. They used an economic model based on the service level agreement to reduce costs and reserve only the necessary CPUs.

In this article, we have combined both approaches, the scheduling in the heterogeneous computing system and the containerization system. First, by designing an efficient scheduler for improving the performance of heterogeneous computing, and second, by applying this scheduler to a popular container orchestrator: Kubernetes. Furthermore, our scheduler not only performs its task when a new job arrives by using the cluster information and historical data, but also whenever there is any change in the state of the cluster (e.g., when an application finishes, a new node appears, and so forth).

4 | JOB EXECUTION MODEL

The deployment of a Job, which usually creates one or more Pods, in a Kubernetes cluster is handled in three main phases, as seen in Figure 1. The first phase is called "Pending phase" (T_{Pen}). Starts when the user submits the Pod, the scheduler detects it and selects the suitable node. We can differentiate two subphases: (i) the period between the submission of the Job and the start of the scheduling process (T_{Wait}) and (ii) the period until the Pod is assigned to one (or more) nodes, called scheduling period (T_{Sch}). The second phase is called "Running phase" (T_{Run}) or "Failed," in the case of an error. It starts when the Pod is assigned to one node and finishes at the end of the task. We can define two periods inside this phase: the time to download the Docker Image and create the container (T_{Cr}) and the execution time (T_{Ex}). The third phase is "Deletion phase" (T_{Del}). Starts when the node finishes the task. The default deletion period in Kubernetes is 30 s.

The following equations show the relations between times:

$$T_{\text{Pen}} = T_{\text{Wait}} + T_{\text{Sch}}, \tag{1}$$

$$T_{\text{Run}} = T_{\text{Cr}} + T_{\text{Ex}}, \tag{2}$$

$$T_P = T_{\text{Pen}} + T_{\text{Run}} + T_{\text{Del}}. \tag{3}$$

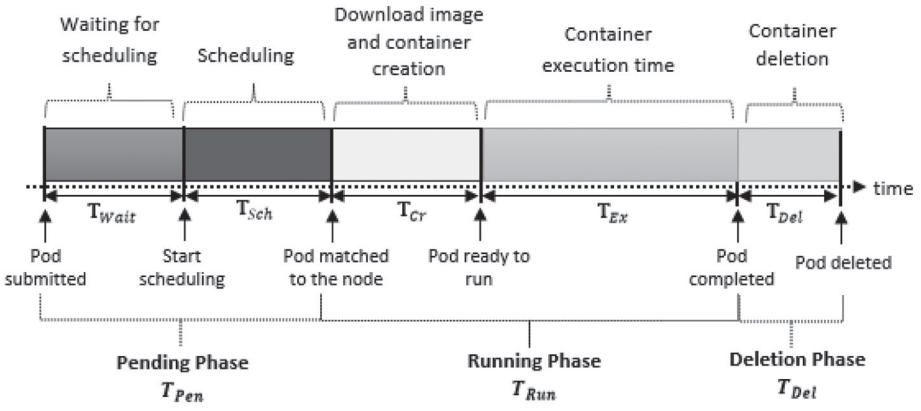


FIGURE 1 Pod lifecycle

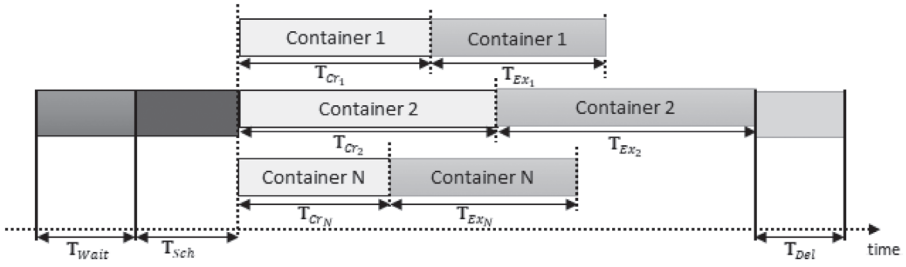
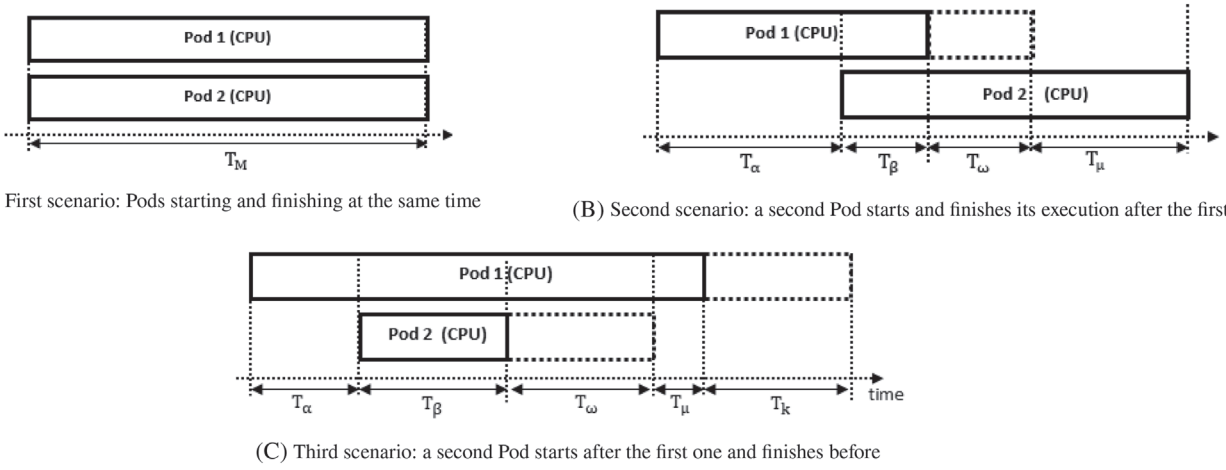


FIGURE 2 Multicontainer Pod lifecycle



(A) First scenario: Pods starting and finishing at the same time

(B) Second scenario: a second Pod starts and finishes its execution after the first one

(C) Third scenario: a second Pod starts after the first one and finishes before

FIGURE 3 Scenarios for multiple Pods in the same node

In order to run a Pod inside a node, we have two scenarios according to the number of containers included in the Pod:

- The Pod includes only one container: it runs after that container is created successfully.
- The Pod includes more than one container: it runs after all the containers are created successfully. Figure 2 depicts the lifecycle of a Pod which has N containers where T_{Cr_1}, T_{Cr_2}, T_{Cr_N}, T_{Ex_1}, T_{Ex_2}, and T_{Ex_N} are the container creation and container execution duration for container 1, container 2, and container N, respectively.

4.1 | Multiple Pods in the same node

Running multiple Pods in the same node may lead to an increase of the execution time of all the Pods on that node, as the resources are shared among them. We call Θ_P to the factor of reduction of the execution speed of the Pod, P is the number of Pods that run simultaneously and τ the time to complete the execution of a Pod in a node. Θ_P depends on the number of Pods P running in the same node, and $P \in \mathbb{N}^*$. The reduction factor affects only the running phase, because the other phases are managed by the master node. Depending on the period of time when the different Pods are executed concurrently in a node, we have three different scenarios presented below.

In the first scenario, two Pods or more start and finish running in the same period of time, as shown in Figure 3(A). The new duration $T_{\mathrm{Run}'}$ is represented in Equation (4).

$$T_{\mathrm{Run}'} = \begin{cases} \tau, P = 1 \\ \tau \times \Theta_P, P > 1. \end{cases} \tag{4}$$

In the second scenario, the second Pod starts and finishes its execution after the first one, as shown in Figure 3(B), where T_α is the amount of time between the start of the first Pod and the second one, T_β and T_ω is the period of time

in which both Pods are executed concurrently, and T_μ the remaining period until the second Pod finishes the execution. The dashed area represents the extra time that the Pod has to run due to sharing resources with other Pods. $T_{\text{Run}'_1}$ in Equation (5) and $T_{\text{Run}'_2}$ in Equation (7) represent, respectively, the total duration of the running time of the first and the second Pod. Moreover, Equations (6) and (8) represent, respectively, the increase in the running time of Pod 1 and Pod 2 due to concurrency.

$$T_{\text{Run}'_1} = T_\alpha + T_\beta + T_\omega, \tag{5}$$

$$T_\beta \times \Theta_P = T_\beta + T_\omega, \tag{6}$$

$$T_{\text{Run}'_2} = T_\beta + T_\omega + T_\mu, \tag{7}$$

$$(T_\beta + T_\omega) \times \Theta_P = T_\beta + T_\omega + T_\mu. \tag{8}$$

In the third scenario the second Pod starts after the first one and finishes before it completes its execution, as shown in Figure 3(C). T_α is the period of time between the start of the first and second Pod, T_β and T_ω is the period of time in which both Pods are executed concurrently, and T_μ and T_k is the remaining period until the first Pod completes its execution. $T_{\text{Run}'_1}$ in Equation (9) and $T_{\text{Run}'_2}$ in Equation (11) represent, respectively, the total duration of the running time of the first and the second Pod. In addition, Equations (10) and (12) represent, respectively, the increase in the running time of Pods 1 and Pod 2 due to concurrency.

$$T_{\text{Run}'_1} = T_\alpha + T_\beta + T_\omega + T_\mu + T_k, \tag{9}$$

$$(T_\beta + T_\omega) \times \Theta_P = T_\beta + T_\omega + T_k, \tag{10}$$

$$T_{\text{Run}'_2} = T_\beta + T_\omega, \tag{11}$$

$$T_\beta \times \Theta_P = T_\beta + T_\omega. \tag{12}$$

4.2 | Using nodes with a GPU

Since Kubernetes does not support sharing a single GPU among different Pods simultaneously[6], when a new Pod that uses a GPU is created (Figure 4), it needs to wait until the Pod that is already using the GPU finishes its task and then it will be created and executed ("running phase").

Therefore, we will have to wait until a GPU is available ($T_{w_{\text{GPU}}}$), as expressed in Equation (13). This time can be further divided in two parts (Figure 5): the time required by the Pod that is running (T_{rem}) and the time required by other queued Pods with higher priority (T_{queue}). $T_{w_{\text{GPU}}}$ is also part of the "pending time" (Equation (14)), so the time to complete the task can be expressed as shown in Equation (15).

$$T_{w_{\text{GPU}}} = \begin{cases} 0, \text{The GPU node is available} \\ T_{\text{rem}} + T_{\text{queue}}, \text{Otherwise} \end{cases}, \tag{13}$$

$$T'_{\text{Pen}} = T_{\text{Pen}} + T_{w_{\text{GPU}}}, \tag{14}$$

$$T_{\text{GPU}} = T'_{\text{Pen}} + T_{\text{Run}} + T_{\text{Del}}, \tag{15}$$

All the variables presented in the previous equations are used in our scheduler to improve the global performance of the system. If the GPU is occupied with one task, the decision unit (DU) of our scheduler (described later in Section 5)

[6]https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

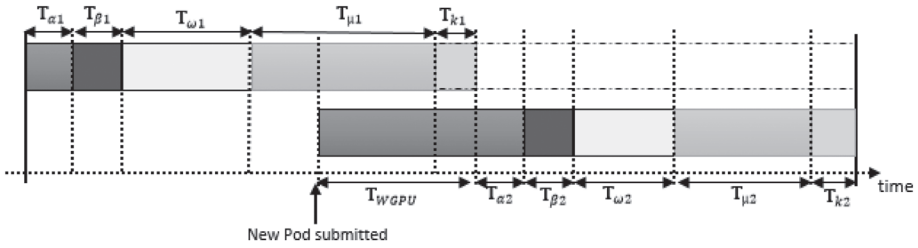


FIGURE 4 Lifecycle of a Pod using a GPU. GPU, graphics processing unit

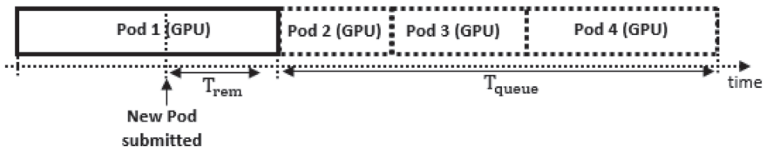
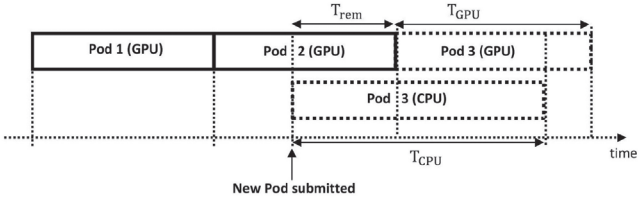
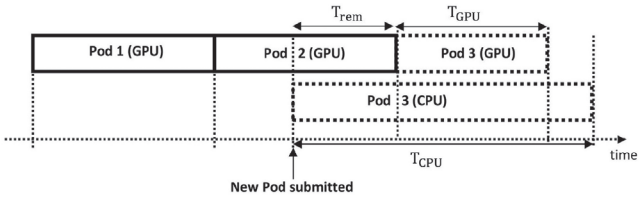


FIGURE 5 The GPU waiting time. GPU, graphics processing unit



(A) The task finishes before if the node with just a CPU is used instead of waiting until the node with the GPU is available



(B) The task finishes before if we wait until the node with the GPU is available

FIGURE 6 Different scenarios for the execution of a task

checks if it is better to assign the Pod to a node without GPU (Figure 6(A)) or if is better to wait until the node with the GPU is available (Figure 6(B)), and then assigns the Pod to the node that is expected to finish first.

5 | ARCHITECTURE OF THE SYSTEM

KubCG is a scheduling system for Kubernetes composed of three main elements: the "DU," the "switch," and the "database."

The DU is the core element. It selects the node that is going to be used to execute a Pod. The switch selects the Docker Image that is suitable for the selected node as the same application may have different implementations adapted to the hardware of the different nodes, or just an implementation that is (for example) both compatible with CPUs and GPUs (many libraries, such as TensorFlow [7] have both CPU and GPU kernels that can be selected at runtime). Finally, the database keeps all the information and statistics related to the previous executions of a Pod.

The work-flow of the system, as seen in Figure 7, is the following:

- A Pod is created in the cluster.
- The DU receives the new Pod.
- The DU sends a request to the Kubernetes API to get the cluster status (Pods and nodes status) and a request to the database server to get information about previous executions of the Pod.
- The DU selects the node that should be used to deploy the Pod, using the historic information recorded in the database, to complete the execution of the Pod as early as possible.
- The switch selects the Docker Image that is compatible with the selected node.
- The Pod is deployed in the node.

[7] https://www.tensorflow.org/

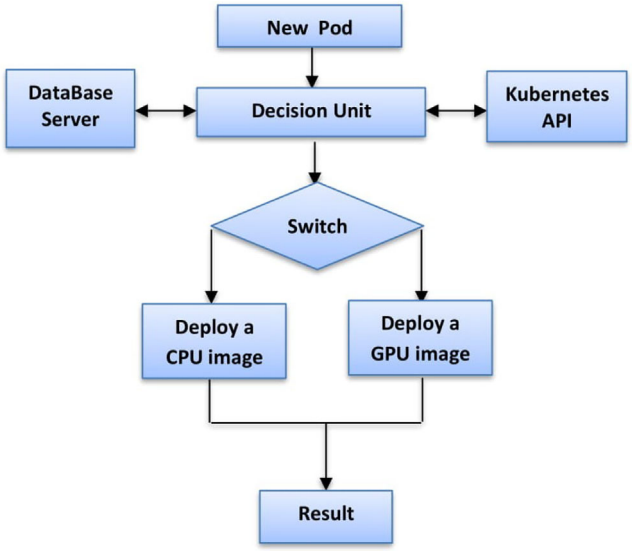


FIGURE 7 Workflow of the system [Color figure can be viewed at wileyonlinelibrary.com]

- After the Pod finishes the task, the data base communicator (DB communicator) sends the information about this execution to be recorded on the database.

We analyse the three elements of the architecture in the following sections.

5.1 | Switch

The switch is responsible for selecting the Docker Image type adapted to the hardware of the node (GPU or plain CPU image). This feature is implemented with a simple table that refers to a container image repository, such as in a Docker hub[8].

The switch receives the node type (CPU or GPU) chosen by the DU and the identity of the Pod, then it searches among the different Docker Image versions and selects the most adequate for the node selected by the DU. Nevertheless, as noted before, a unique container Image may be compatible with different nodes. After that, the switch asks the images Registry to deploy the selected image to the node. Algorithm 1 describes the switch mechanism (IMGP is the image associated to the pending Pod).

Algorithm 1. Selecting a Docker Image

Input : Resource Type, Image Pending
Output: Image type
if *Resource* = CPU **then**
  IMGD=ImageTable$[IMGP]_{CPU}$ ▷ Select a Docker Image designed for being executed in a CPU
else
  IMGD=ImageTable$[IMGP]_{GPU}$ ▷ Select a Docker Image designed for being executed in a GPU
end

5.2 | Database

In order to record, analyse and use the information regarding the execution of previous instances of the Pod, the system saves different statistics in a database which includes four tables.

[8]https://hub.docker.com

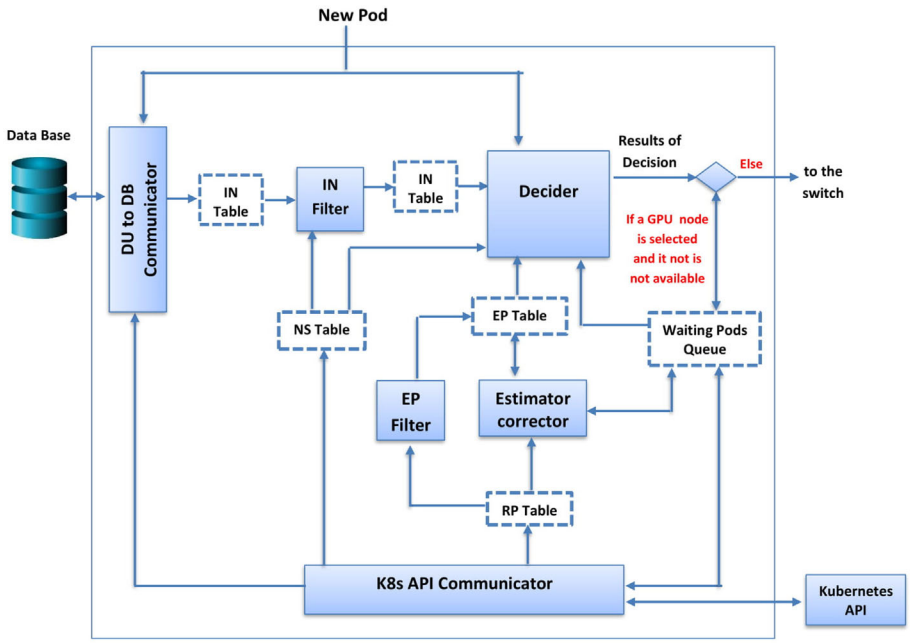


FIGURE 8 Decision unit [Color figure can be viewed at wileyonlinelibrary.com]

- Images: includes the names of the available images.
- Nodes: information about the nodes available in the cluster.
- Records: results recorded after the execution of each Pod.
- Statistics: average, minimum, and maximum execution times for each Pod.

All this new information would be collected and updated after the addition of new images or resources.

5.3 | Decision unit

The DU selects the node that is expected to complete the execution of the Pod first. In order to make the choice, the DU receives information about the availability of resources in the different nodes (e.g., the availability or not of a GPU) and their state (e.g., the Pods that are using that node), the statistics about the previous execution of the Pod that is going to be deployed, and the compatibility between the Pod and the node (e.g., if the node can run the libraries or software included in the Pod). For sake of simplicity, in this description we will consider only nodes with a CPU or a CPU and a GPU.

5.3.1 | DU components

The DU has the following components (Figure 8):

1. DU to Database communicator: It is responsible to send information about the request and receive the data from the database server.
2. Image per node table (IN table): Stores the information received from the database, which includes all the information related to the Docker Image associated with new Pod.
3. Image per node filter (IN filter): Removes all the columns without active nodes in IN table.
4. Node status table (NS table): Stores the current status of each node.
5. Decider: Selects the best node and resource type to run the new Pod.
6. Estimator corrector: Controls and corrects the time that was estimated for the completion of the running Pods.
7. K8s API communicator: Uses the Kubernetes API to know the status of the nodes and the Pods they are executing.
8. Running Pod table (RP table): Stores information about the Pods running in Kubernetes.
9. Estimated Pod table (EP table): Stores the time that is estimated for the completion of the execution of the Pods.

10. Waiting Pods queue: Stores the information related to the Pods that are waiting for a GPU.
11. Estimated Pod filter (EP filter): Filters the information in EP table related to Pods that have already finished.

5.3.2 | DU algorithm

Algorithm 2. The estimator corrector

Input : $ET_P, AVG_{ED}, MIN_{ED}, MAX_{ED}$
Output: ED_N
if $CT > ET_P$ ▷ *Check if CT exceeded the* ET_P
then
  if *previous technique is MIN* **then**
    $ED_N = AVG_{ED}$ ▷ Modify the estimated duration from MIN_{ED} to the AVG_{ED}
  else if *previous technique is AVG* **then**
    $ED_N = MAX_{ED}$ ▷ Modify the estimated duration from AVG_{ED} to the MAX_{ED}
  else
    $ED_N = ED_P + MIN_{ED}$ ▷ If all the previous techniques were already used, MIN_{ED}
    ▷ duration is added each time the algorithm is executed
  end
end

The DU follow the next steps to choose the best node for the new Pod:

1. K8s API Communicator sends a request to the Kubernetes Master API to receive the current state of the nodes and the Pods. Then, it saves the information in the NS table and RP table, respectively.
2. The EP filter receives the currently running Pods from the K8s communicator. Next, it checks by using the RP table if the Pods in the EP table are still running in the cluster or not. Then, it removes from the EP table the Pods which are not in the RP table received.
3. Once the EP table is updated, the Estimator corrector receives it from the EP filter and by using Algorithm 2, it compares the previous estimated time, ET_P, and the current time, CT. In case CT exceeded ET_P, the algorithm follows the next steps to adjust the estimated duration (ED):
   - If the previous technique used is the minimum estimated duration MIN_{ED}, the algorithm uses now the average estimated duration AVG_{ED}.
   - If the previous technique used is the average estimated duration AVG_{ED}, the algorithm uses now the maximum estimated duration MAX_{ED}.
   - If all the techniques were used, the algorithm adds the MIN_{ED} in each verification.

   Furthermore, the Estimator corrector updates the waiting Pod table for each Pod that was waiting for a node that changed its state. Finally, the estimator corrector reports the new estimated duration ED_N to the decider.
4. The DU to DataBase Communicator sends a request associated with the Docker Image name of the new Pod, denominated IMG_N, to the data base server.
5. The DataBase Communicator receives the data from the database server (all the recorded results in the statistics table that have the same IMG_N) and saves them on the image per node (IN) table.
6. The DataBase Communicator sends the IN table to the IN filter to remove the information related to the inactive nodes, with the information stored in the NS table.
7. Once the IN table is updated, the decider receives the IN table from the IN filter, and the EP table and waiting Pods queue information that were previously updated by the Estimator corrector. There are different potential situations according to the information stored in the IN table:

- The IN table does not have any information because that IMG_N was not deployed before. The decider makes the decision depending on the current situation of the nodes:
  - If all the nodes have at least one Pod running, the decider checks which node is going to reach an idle state first (i.e., not running any Pod) and selects it.
  - If only one node is in idle state, the decider selects it.
  - If more than one node are idle or more than one node reach the idle state at the same time the decider sends an other request to the database through the DU to DB Communicator in order to verify which node will complete the task before.
- There is information about the performance of $IMG_N in all the nodes and resources types. In this case, the Decider categorizes the Pods according to the information that they have to process:
  - In case the Pod has to process an input file (e.g., a video processing task may be provided with a video file), the Decider uses information about the previous executions of the same Pod with similar files to estimate the duration of the task in each node and selects the best node (Algorithm 3).
  - In case there is no an input file, the Decider selects directly the best node by following the Algorithm 3.

8. When a Pod finishes the task successfully, the DU to DB Communicator receives a notification from the K8s API Communicator to send the information related to the execution of that Pod to the database server.

In addition, when a GPU node becomes available, the DU sends the Pod at the beginning of the queue to the switch.

Algorithm 3. Selection of the best node and resource

Input : INtable
Output: Best node and resource
$\text{Min}_{\text{GPU}} = \text{Min}(\Delta_{\text{GPU}_i \in [1..G]})$ ▷ Select the GPU node with the best performance
$\text{Min}_{\text{CPU}} = \text{Min}(\Delta_{\text{CPU}_i \in [1..C]})$ ▷ Select the CPU node with the best performance
if $\text{Min}_{\text{GPU}} > \text{Min}_{\text{CPU}}$ **then**
  Best node = GPU_i ▷ Assign the node *i* to the best node
else
  Best node = CPU_i ▷ Assign the node *i* to the best node
end

The selection algorithm

Algorithm 3 describes how the system decides which node is the best for the deployment of a new Pod. This algorithm divides the IN table in two sections, for nodes without and with GPU. For nodes without GPU, the algorithm estimates the duration of the task as Δ_{CPU_i}, being i the id of the node. As shown in Equation (16), the assigned value is the minimum running time ($Min($T_{Run}$)$) recorded in the statistics table when the CPU node is idle, or $T_{\text{Run}'}$ (defined by Equation (4)) in other case.

$$\Delta_{\text{CPU}_i} = \begin{cases} Min(T_{\text{Run}_i}), & \text{if the node } i \text{ is idle} \\ Min(T_{\text{Run}'_i}), & \text{if the node } i \text{ is not idle} \end{cases}. \tag{16}$$

For nodes with a GPU, the Decider estimates the time that the Pod has to wait before being scheduled ($T_{w_{\text{GPU}}}$, Equation (13)). This value will include also the time to execute other Pods that have a higher priority (Figure 17).

$$\Delta_{\text{GPU}_i} = Min(T_{\text{Run}_j}) + T_{w_{\text{GPU}}} + T_{\text{queue}}. \tag{17}$$

The algorithm selects the node that will complete the task before.

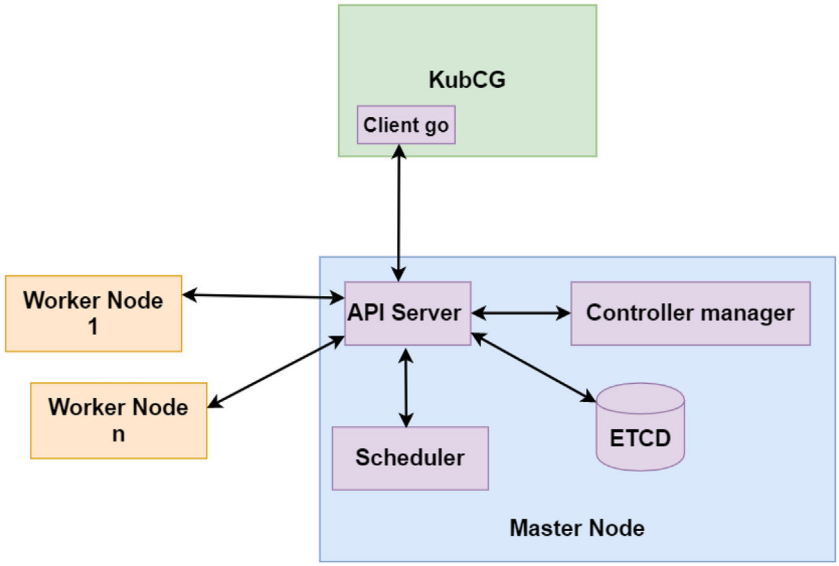


FIGURE 9 Architecture of the implementation [Color figure can be viewed at wileyonlinelibrary.com]

TABLE 1 Hardware used in the experiments

Node	Node type	CPU	GPU	Memory
Node 00	Master	8 * Intel Xeon E5620	–	8 GB
Node 01	Worker	16 * Intel Xeon E5520	–	12 GB
Node 02	Worker	16 * Intel Xeon E5520	–	12 GB
Node 03	Worker	8 * Intel Corei7-4790	1 * GeForce GT 730 2 GB	16 GB
Node 04	Worker	8 * Intel Core i7-6700	1 * GeForce GTX 1050 Ti 4 GB	16 GB
Node 05	Worker	8 * Intel Corei7-4790	1 * GeForce GTX 1050 Ti 4 GB	16 GB

Abbreviation: GPU, graphics processing unit.

6 | IMPLEMENTATION AND EXPERIMENTS

The proposed custom scheduler, KubCG, was implemented and tested through a set of experiments in order to validate its operation and to compare its performance with the default Kubernetes scheduler. KubCG was implemented taking advantage of the architecture and tools provided by Kubernetes[9]. It is possible to completely replace the default scheduler by modifying the original code and recompiling it, implement an extension called by the default scheduler when making scheduling decisions, or package a custom scheduler binary into a container image and deploy it in the Kubernetes cluster. We followed the latter approach because of advantages such as not having to modify the default Kubernetes setup to use our scheduler. Nevertheless, in order to use KubCG for a Pod, its name has to be supplied as a value to the name `spec.schedulerName` in the Pod definition. Figure 9 shows the architecture of the implementation. KubCG was implemented in Go and uses the client-golibrary to interact with the API Server for tasks such as discovering the newly created Pods, obtaining the list of nodes and their status, or assigning pods to the desired nodes.

6.1 | Experiments

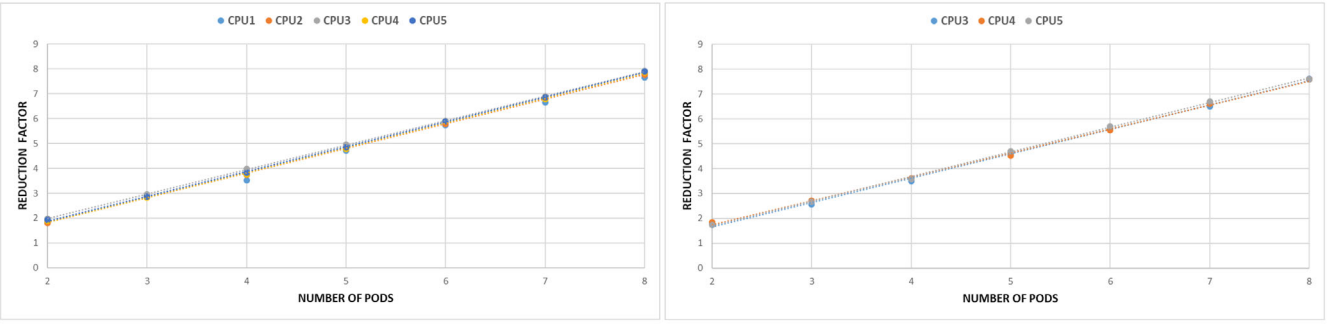
We deployed a Kubernetes cluster with one master node and five worker nodes with different capabilities. Table 1 describes the characteristics of the machines used in the cluster. We label as CPU1, CPU2, CPU3, CPU4, and CPU5 the CPU resources in nodes from 01 to 05 and GPU3, GPU4, and GPU5 the GPU resources in nodes 03, 04, and 05.

[9]https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/

TABLE 2 Description of the Docker images used in the experiments

Image	Image type	CPU usage	GPU usage	Description
Scale	CPU	550%	0%	Scale video with ffmpeg
	GPU	34%	20%	Scale video with ffmpeg and CUDA
Encode	CPU	705 %	0%	Encode raw video to H264 with ffmpeg
	GPU	36%	33%	Encode raw video to H264 with ffmpeg and CUDA
Matrix	CPU	102%	0%	Matrix multiplication (coded in C)
	GPU	48%	100%	Matrix multiplication (coded in C and CUDA library)
Tensorflow	CPU	612%	0%	Tensorflow training job for an image classifier
	GPU	77%	82%	Tensorflow training job for an image classifier

Abbreviation: GPU, graphics processing unit.



(A) Reduction factor with several video encoding Pods

(B) Reduction factor with several Tensorflow Pods

FIGURE 10 Reduction factor [Color figure can be viewed at wileyonlinelibrary.com]

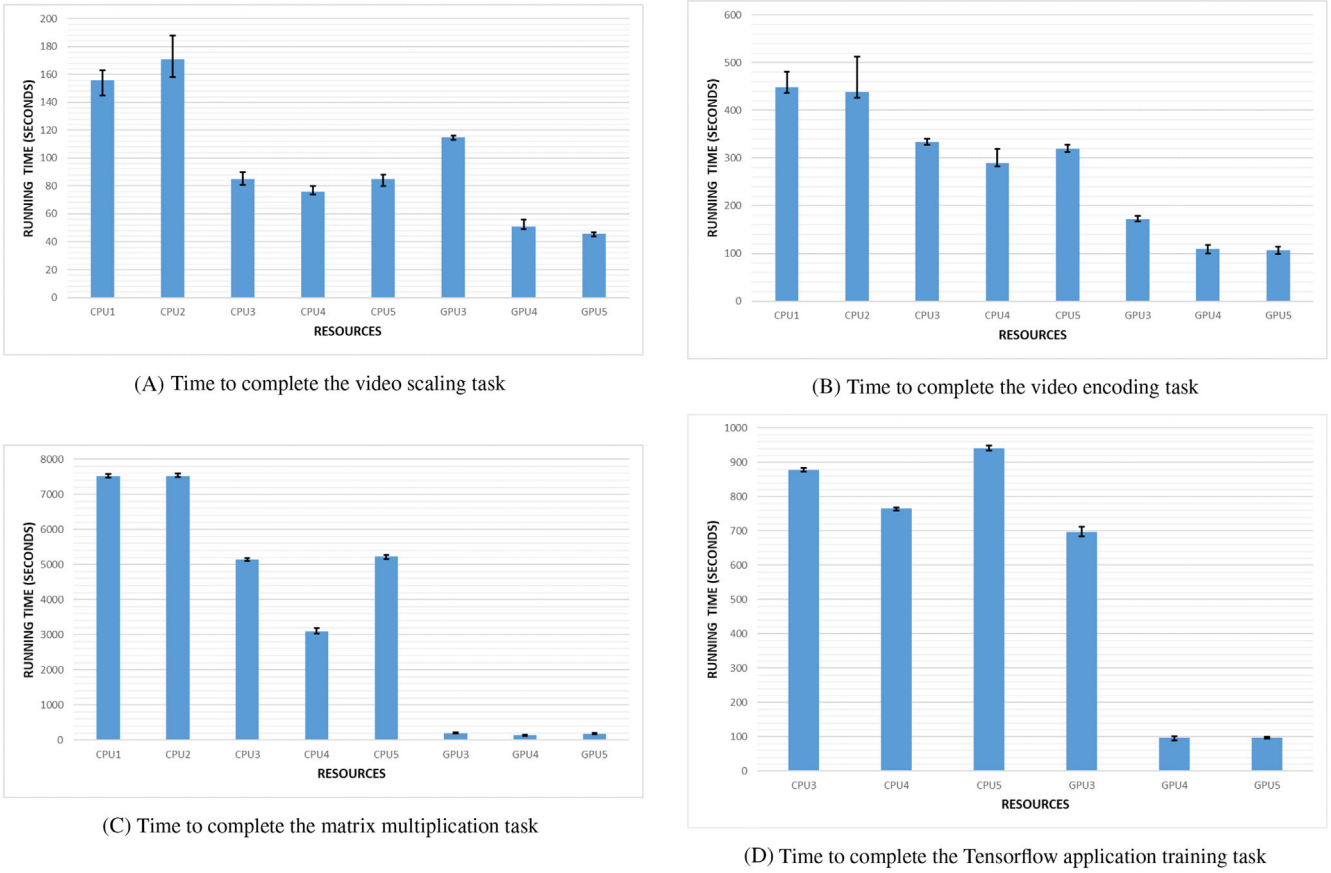
The nodes operating system was Ubuntu 16.04 and Kubernetes 1.13 was installed. The machines that have a GPU also were configured with the NVIDIA driver version 410 and CUDA 10.

For the experiments, we created two image types: one intended for nodes with just a CPU and other for nodes with a GPU compatible with CUDA. Table 2 provides more details about the different images that were used to benchmark the implementation, as well as the use of CPU and GPU when running these images. Note than a CPU usage greater than 100% means that multiple cores where used. In addition, we measured the execution time in seconds and, in order to simplify the experiments related to video processing, we used the same video file.

We completed four sets of experiments. The first one comparing how KubCG and the default Kubernetes scheduler distributed the workload among the cluster nodes in a static scenario. The second one performed the same analysis considering the random creation of Pods following a Poisson process. Then, we deployed multiple Pods simultaneously and measured the time until all the tasks were completed. Next, we studied the scenario where multiple Pods are launched randomly in a time period. Finally, we repeated the previous test by using different types of images at the same time.

The measured time in the experiments was the running time T_{Run}. However, when a GPU node was used, we considered also the pending time T'_{Pen} described in Section 4.2 to show the effect of using the model presented in Figure 6(A,B).

Regarding to the reduction factor (Θ_P) that represents how the time to complete a task increases by executing several Pods in parallel, as described in Section 4.1, we measured its value by running multiple video encoding and multiple Tensorflow images in the same node. The results are shown in Figure 10 (note that in this particular scenario the Tensorflow image is not compatible with the CPU1 and CPU2). The results show how the time to complete a task increases linearly with the number of Pods running simultaneously. From the results of this experiment we can approximate $\Theta_P = P$ where $P \in N^*$ is the number of Pods executed simultaneously.



(A) Time to complete the video scaling task

(B) Time to complete the video encoding task

(C) Time to complete the matrix multiplication task

(D) Time to complete the Tensorflow application training task

FIGURE 11 Average, maximum, and minimum time to complete different tasks in each node [Color figure can be viewed at wileyonlinelibrary.com]

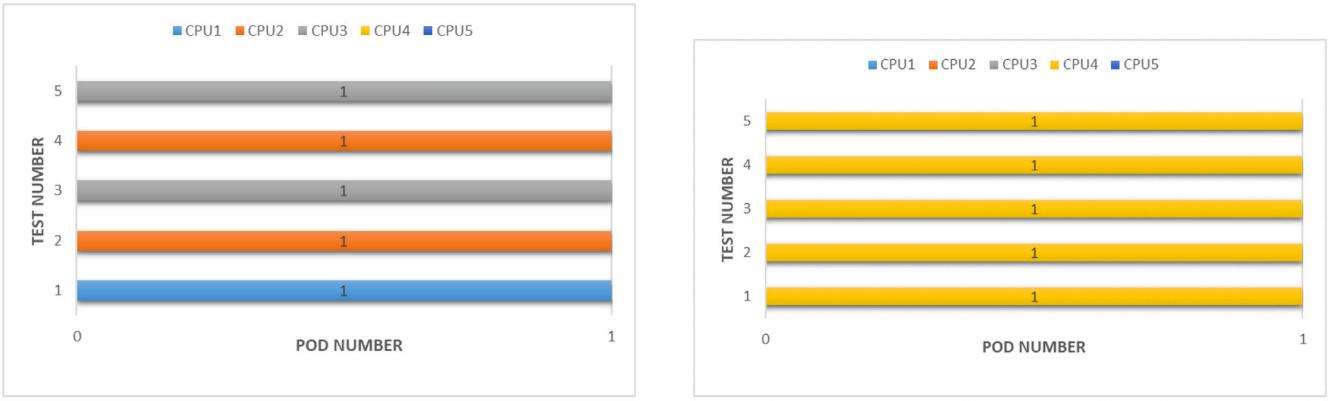
KubCG requires historic information about the executions of the different Pods. In order to validate its operation we selected a set of applications (Pods), we deployed 100 times each Pod and we stored the execution statistics in the database, which includes an entry for each node and Pod with the maximum, minimum and average execution times.

The results obtained for each node are shown in Figure 11(A) (video scaling), Figure 11(B) (video encoding), Figure 11(C) (matrix multiplication), and Figure 11(D) (Tensorflow training). The bar represents the average value, while the range in black shows the minimum and maximum times saved in the database. Note that our KubCG scheduler detected that the Tensorflow image is not compatible with CPU1 and CPU2, that means that the scheduler avoided automatically deploying that image in those resources, that is, the reason why those CPUs are not shown in Figure 11(D).

Figure 11 also differentiates the time to complete the execution using only the CPU or the CPU and the GPU in nodes 3, 4, and 5. Almost all of the results show how GPUs outperform CPUs for the tested tasks. However, the speed-up between the CPU time, T_{CPU}, and the GPU time, T_{GPU}, calculated as the ratio T_{CPU}/T_{GPU}, varies among the images, for instance, the speed-up of CPU5 and GPU5 with the scale Docker Image is 1.85 and the speed-up of CPU5 and GPU5 with the matrix Docker Image is 28.4. The results also show that CPU3, CPU4, and CPU5 provide a lower running time than GPU3 for video scaling.

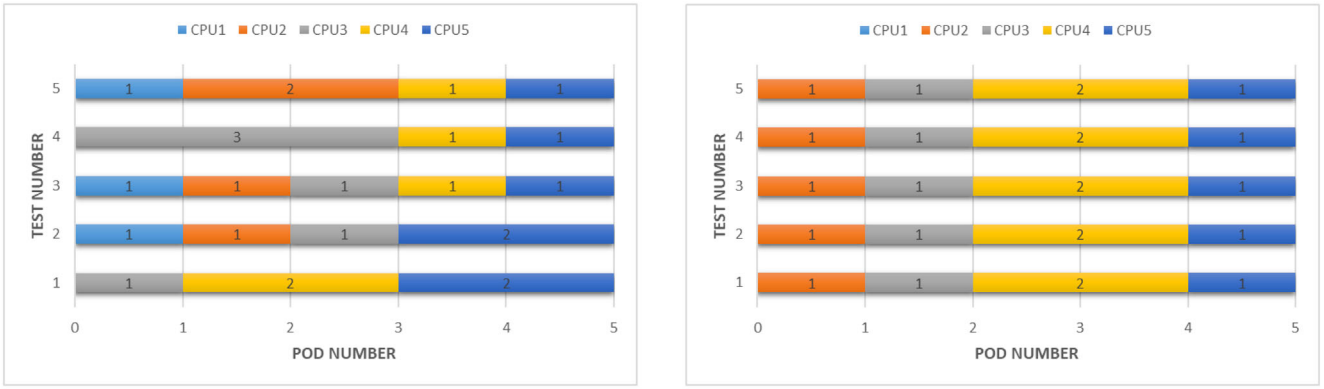
6.2 | Workload distribution

In order to validate our scheduler we employed the same set of applications used to generate the historic information. We deployed the same Pod five times in sequence and we checked which node was selected by the scheduler to execute the Pod. Next, we repeated the same test by deploying the five Pods in parallel in five different times and we noted the node used. Figures 12, 13, 14, and 15 show the results for the scale and matrix images using the default scheduler and KubCG. The number inside the bars represents the number of Pod deployed in that node.



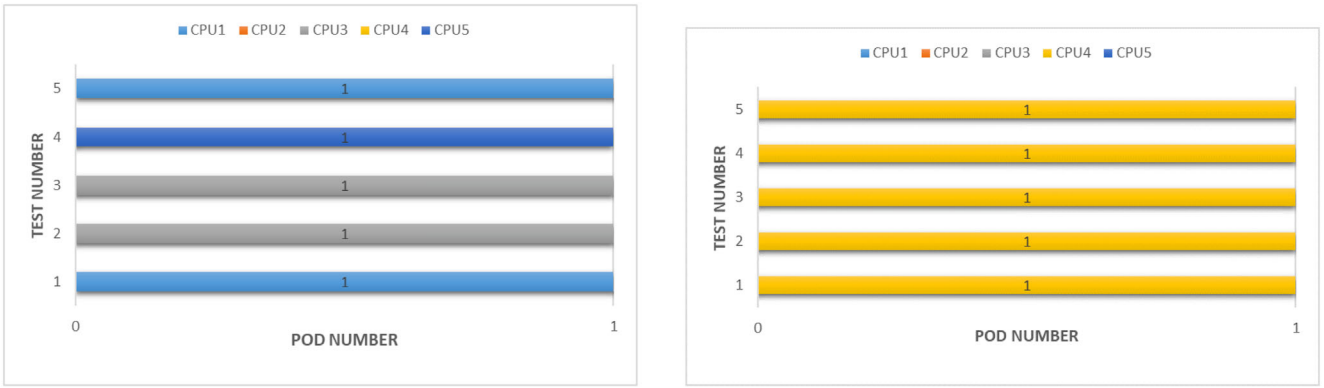
(A) Default (B) KubCG

FIGURE 12 Video scaling Pod assignation (five sequential times) [Color figure can be viewed at wileyonlinelibrary.com]



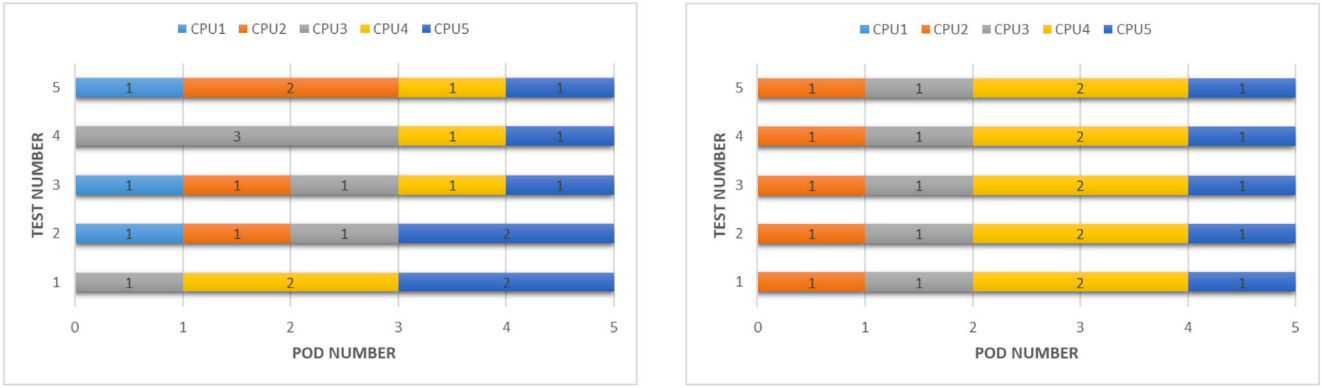
(A) Default (B) KubCG

FIGURE 13 Video scaling Pod assignation (five Pods in parallel, five sequential times) [Color figure can be viewed at wileyonlinelibrary.com]



(A) Default (B) KubCG

FIGURE 14 Matrix multiplication Pod assignation (five sequential times) [Color figure can be viewed at wileyonlinelibrary.com]



(A) Default (B) KubCG

FIGURE 15 Matrix multiplication Pod assignation (five Pods in parallel, five sequential times) [Color figure can be viewed at wileyonlinelibrary.com]

We can observe that our KubCG scheduler assigns the Pods to the same nodes in all the consecutive tests because it uses the statistics recorded on the database, so it can evaluate where the Pod should be deployed to obtain the best results. However, with the default scheduler, the assigned node varies. This explains why in the next experiments the results of default scheduler has a higher minimum and maximum range than our scheduler.

6.3 | Random creation of Pods following a Poisson process

In this experiment, we modeled an scenario where Pods are created randomly following a Poisson process, where λ is the rate parameter (Equation (18)). We choose five values for λ: 3, 5, 10, 20, and 50, which means that the average numbers of Pod creation every 10 min are 3, 5, 10, 20, and 50. We repeat the experiments 10 times and we calculate the average results.

$$F_x = 1 - e^{-\lambda x}. \tag{18}$$

Figure 16 represents the average duration of Pods running. D-CPU and D-GPU show the results for the default scheduler using CPU and GPU images, respectively, and K-CPU and K-CPU+GPU the results for KubCG. The bar represents the average time to complete each task, and the dark line shows the minimum and maximum values. Please note that the notation K-CPU+GPU is used to represent the ability of KubCG to switch between CPU and GPU images.

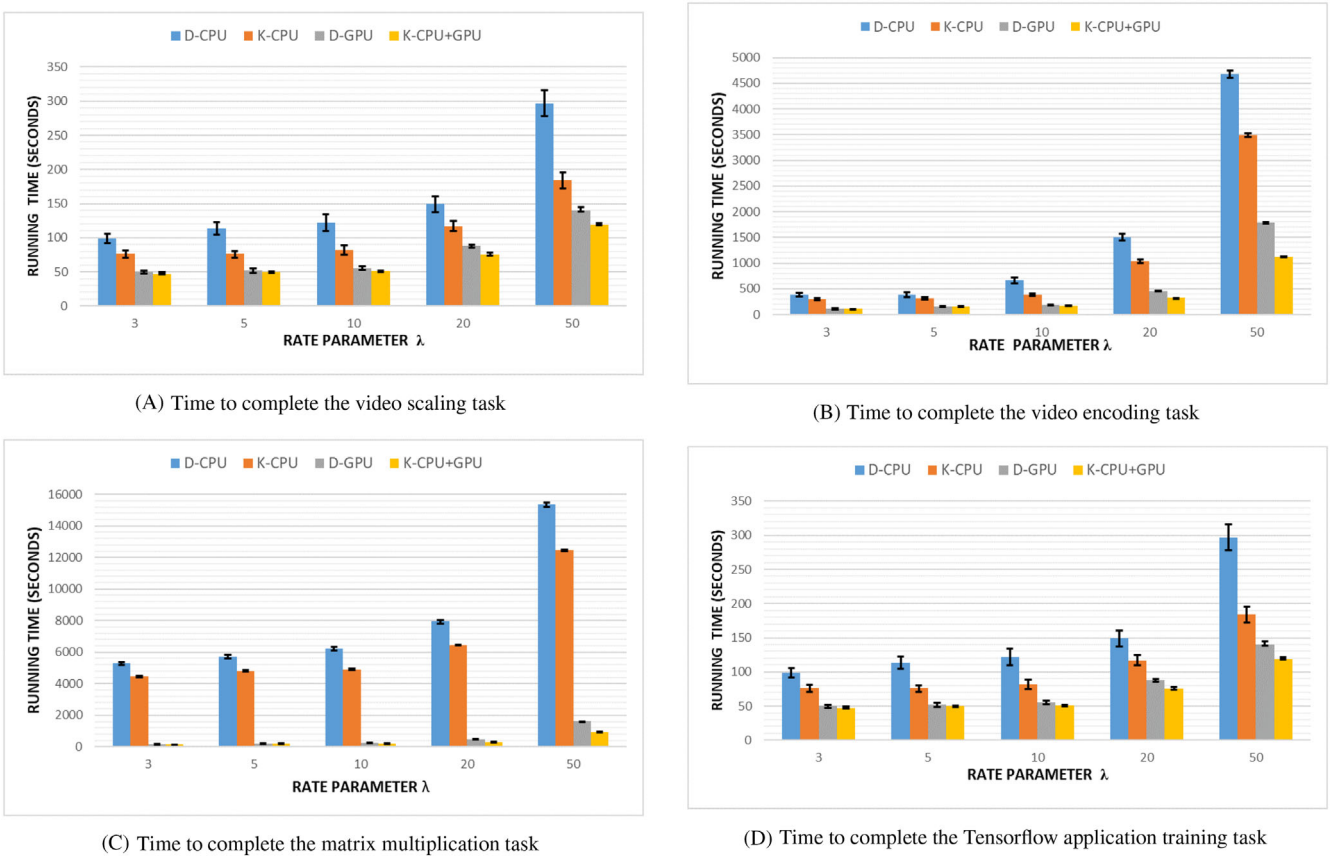
The bar represents the average time to complete each task, and the dark line shows the minimum and maximum values.

In the different tests KubCG obtained better results than the default scheduler. Using only CPU nodes the time to complete the tasks using KubCG is reduced by a 36% (with $\lambda = 3$), 33% (with $\lambda = 5$), 41% (with $\lambda = 10$) , 32% (with $\lambda = 20$), and 38% (with $\lambda = 50$) of the time with the default scheduler.

When using also GPUs, KubCG can improve the performance up to a 8% with $\lambda = 3$, 9% with $\lambda = 5$, 22% with $\lambda = 10$, 39% with $\lambda = 20$, and 44% with $\lambda = 50$. In addition, KubCG is much more consistent, as the difference between the maximum and minimum time to complete the same task in the different experiments is lower.

6.4 | Multiple Pods running concurrently

With these experiments, we tested the deployment of multiple Pods at the same time. We completed five different tests by running 3, 5, 10, 20, and 50 Pods simultaneously. Figure 17 shows the time required to complete all the tasks. The figures show the results for the video scaling, video encoding, matrix multiplication, and tensorflow training Pods, respectively.



(A) Time to complete the video scaling task

(B) Time to complete the video encoding task

(C) Time to complete the matrix multiplication task

(D) Time to complete the Tensorflow application training task

FIGURE 16 Average, maximum, and minimum time to complete different tasks in each node with random arrival times [Color figure can be viewed at wileyonlinelibrary.com]

The vertical axis is the time to complete all the tasks in seconds, and the horizontal axis is the number of Pods running simultaneously in the cluster.

Like with previous tests, KubCG provided better results in this set of experiments than the default scheduler. Using only the nodes without a GPU, the time to complete the tasks with KubCG is reduced up to a 43%, 41%, 48%, 44%, and 47% of the time required by the default scheduler, when the number of Pods is equal to 3, 5, 10, 20, and 50, respectively.

When we compare the time to complete the tasks using the nodes using also the GPUs, KubCG reduces the time up to the 50%, 37%, 37%, 55%, and 64% of the time required with the default scheduler, when the number of Pods is equal to 3, 5, 10, 20, and 50, respectively.

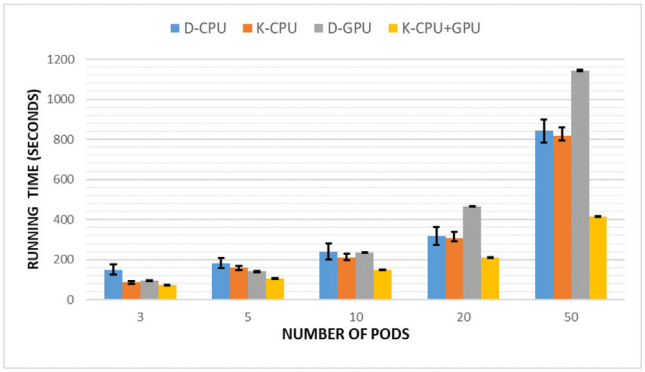
Furthermore, in the video scaling experiment KubCG even obtains better results using just the CPU than the default scheduler using the GPU. This happens because the default scheduler does not consider using a CPU node when all the GPU nodes are occupied.

6.5 | Partial overlapping between Pods

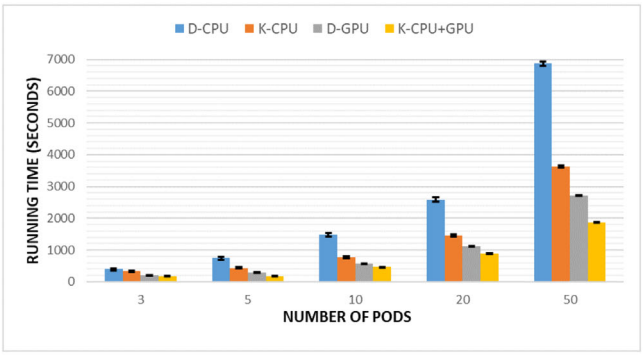
In this part, we completed a test where we forced the overlap between the execution of Pods, as shown in Figure 3(B) (partial overlap of Pods running in CPUs), Figure 6(A) (overlapping between a CPU node and GPU node), and Figure 6(B) (overlap among Pods requesting the GPU nodes).

We completed four different sets of experiments, in which we deployed the same number of Pods randomly (10 instances of video scaling, video encoding, Tensorflow, or matrix multiplication Pods) in three different periods of time: 250 s (I_{250}), 750 s (I_{750}), and 1500 s (I_{1500}).

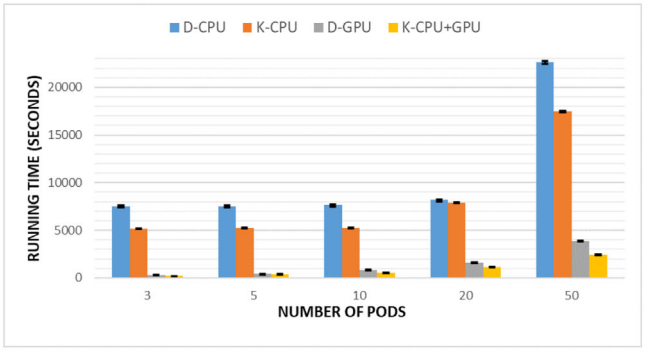
As shown in Figure 18, KubCG obtained better results. Using only CPU nodes the time to complete the tasks using KubCG can be reduced down to be a 53% (with I_{250}), 48% (with I_{750}), or 34% (with I_{1500}) of the time with the default



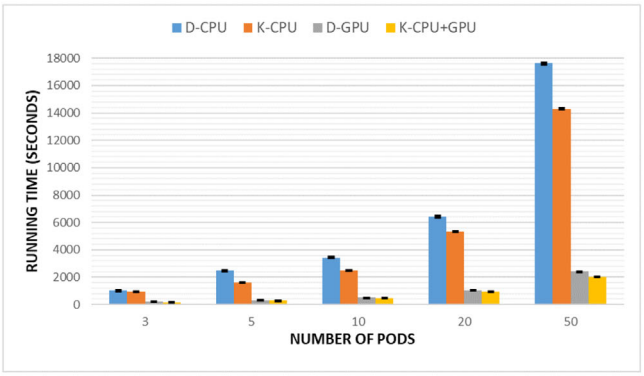
(A) Time to complete multiple video scaling tasks



(B) Time to complete multiple video encoding tasks



(C) Time to complete multiple matrix multiplication tasks



(D) Time to complete multiple Tensorflow application training tasks

FIGURE 17 Average, maximum, and minimum time to complete different tasks in each node running in parallel [Color figure can be viewed at wileyonlinelibrary.com]

scheduler. When using also GPUs, KubCG can improve the performance up to a 34% with I_{250}, 48% with I_{750}, and 42% with I_{1500}.

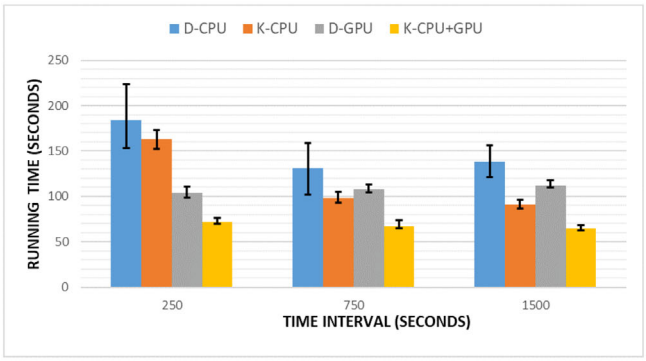
6.6 | Interference between different types of images

In this section, we describe the results of a set of experiments in which we combined images of different types (described in Table 2).

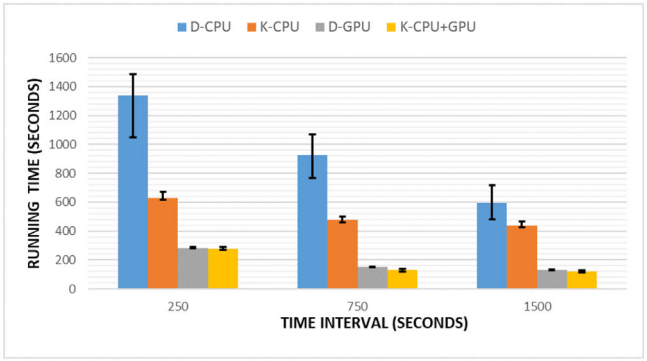
First, we repeated the experiment described in subsection 6.3, but choosing four values for λ: 4, 8, 16, and 32, which means that the average number of Pods created every 10 min is 4, 8, 16, and 32. In each test, we used the same number of images of each type. For example, as we are working with four different types of images, for $\lambda = 8$, we deployed two Pods of each type. As we can see in Figure 19(A), using only the CPU nodes, the time to complete the tasks using KubCG is reduced by a 20% compared with the time needed when using the default scheduler. Furthermore, when using also the GPUs, KubCG can improve the performance up to 33%.

We also tested the deployment of multiple Pods at the same time. We completed four different tests by running 4, 8, 16, and 32 Pods simultaneously, using the same number of images of each type. Figure 19(B) shows the time required to complete all the tasks. The speed-up of KubCG is about 20%, compared with the default scheduler with CPU nodes, and 34% taking also advantage of the GPUs.

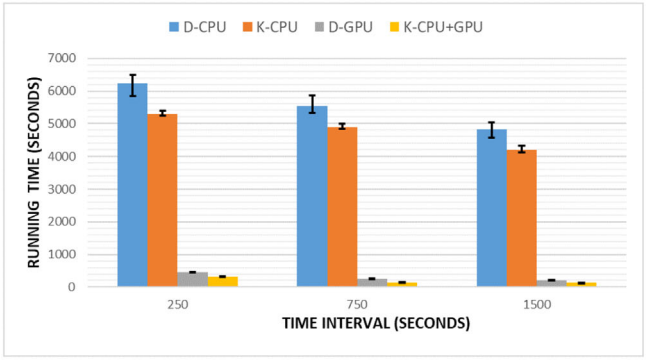
Finally, we completed an experiment in which we deployed the same number of Pods randomly (two instances of video scaling, video encoding, Tensorflow, or matrix multiplication Pods) in three different periods of time: 250 s (I_{250}), 750 s (I_{750}), and 1500 s (I_{1500}). The results are shown in Figure 19(C), where we can see that, when using only CPU nodes, KubCG reduces the running time by an 18% of the time needed with the default scheduler, and up to a 51% when the GPU nodes are enabled.



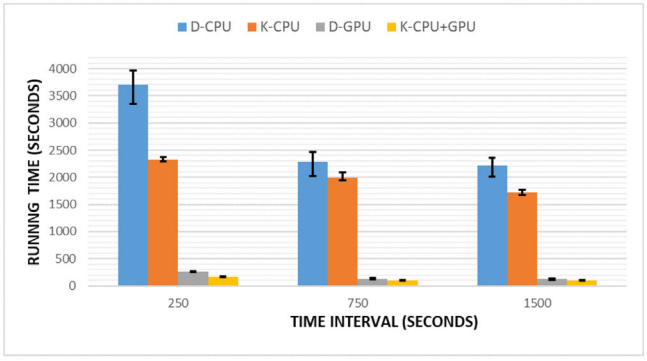
(A) Time to complete the video scaling task



(B) Time to complete the video encoding task

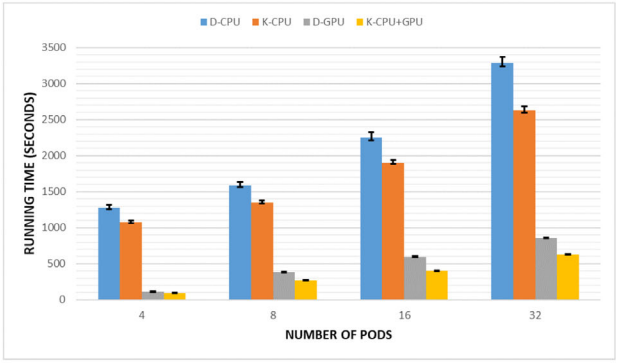


(C) Time to complete the matrix multiplication task

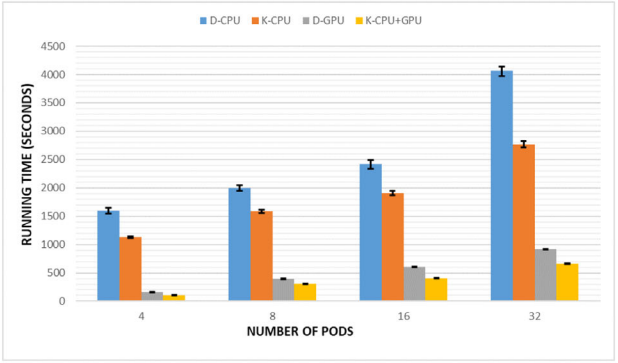


(D) Time to complete the Tensorflow application training task

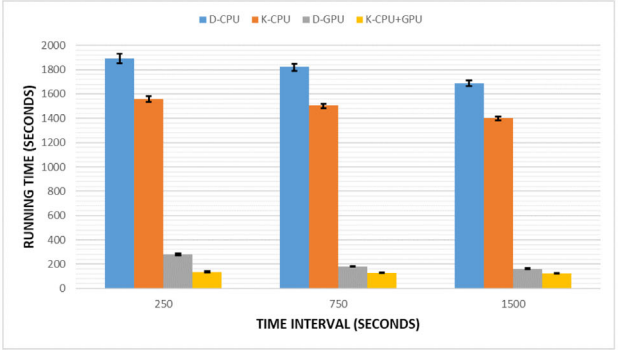
FIGURE 18 Average, maximum, and minimum time to complete different tasks with partial overlapping between Pods [Color figure can be viewed at wileyonlinelibrary.com]



(A) Time to complete various types of task with random arrival times



(B) Time to complete various types of task running in parallel



(C) Time to complete various types of task with partial overlapping between Pods

FIGURE 19 Average, maximum, and minimum time to complete a mixed set of tasks [Color figure can be viewed at wileyonlinelibrary.com]

7 | CONCLUSIONS AND FUTURE WORK

Container-based virtualization provides a mechanism that allows multiple users to share computational resources efficiently. In addition, many applications, such as video processing or machine learning, can take advantage of the use of hardware acceleration devices (principally GPUs) to provide better results. Furthermore, new paradigms such as MEC or the Industry 4.0 are fostering the creation of a large set of small cloud computing infrastructures in telco operator installations or in industries. Each MEC deployment has to be optimized in order to avoid skyrocketing costs, for example, by reducing the number of expensive components (such as GPUs or other specialized acceleration devices). Nevertheless, due to the lack of fractional sharing of a single GPU in containerization environments, GPU demanding tasks executed in such systems could suffer high latency times, which is unacceptable. Thus, it is necessary to create new mechanisms to optimize the usage of the hardware, and we have shown how an efficient scheduler can be a solution to this problem.

We have designed an architecture to manage the deployment of Kubernetes jobs that make intelligent use of nodes and resources, such as CPUs and GPUs, depending on the kind of job that the user wants to deploy. In this article, we have described how our proposal was implemented and the different tests completed. The main component in our proposal is a scheduler that uses information about the previous executions of a particular application to select the best node. This mechanism is specially efficient for applications that can be both executed in a CPU or in a GPU. We compared our scheduler KubCG with the default scheduler used by Kubernetes, and our measurement proved that the running time was reduced down to a 64% of the original time.

As future work, we will explore the usage of machine learning techniques to assign resources to the different Pods, and compare its performance with KubCG and the default scheduler. We will also analyse new mechanisms to implement a fractional GPU sharing system, such as the creation of virtual GPUs, to run multiple applications simultaneously on a single GPU.

ACKNOWLEDGMENT

This work was funded by the European Commission under the Erasmus Mundus E-GOV-TN project (E-GOV-TN Erasmus Mundus Action 2 Lot 6 Project no. 2013-2434).

ORCID

Felipe Gil-Castiñeira https://orcid.org/0000-0002-5164-0855

REFERENCES

1. Mach P, Becvar Z. Mobile edge computing: a survey on architecture and computation offloading. *IEEE Commun Surv Tutor*. 2017;19(3):1628-1656. https://doi.org/10.1109/COMST.2017.2682318.
2. Ananthanarayanan G, Bahl P, Bodík P, et al. Real-time video analytics: the killer app for edge computing. *Computer*. 2017;50(10):58-67. https://doi.org/10.1109/MC.2017.3641638.
3. Bernstein D. Containers and cloud: from LXC to docker to kubernetes. *IEEE Cloud Comput*. 2014;1(3):81-84. https://doi.org/10.1109/MCC.2014.51.
4. Hong C-H, Spence I, Nikolopoulos DS. GPU virtualization and scheduling methods: a comprehensive survey. *ACM Comput Surv*. 2017;50(3):35:1–35:37. https://doi.org/10.1145/3068281.
5. Sanders J, Kandrot E. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st ed. Boston, MA, USA: Addison-Wesley Professional; 2010.
6. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and Linux containers. Paper presented at: Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Philadelphia, PA, USA; 2015:171-172.
7. Kämäräinen T, Shan Y, Siekkinen M, Ylä-Jääski A. Virtual machines vs. containers in cloud gaming systems. Paper presented at: Proceedings of the 2015 International Workshop on Network and Systems Support for Games (NetGames). Zagreb, Croatia; 2015:1-6.
8. Gupta V, Gavrilovska A, Schwan K, et al. GViM: GPU-accelerated virtual machines. Paper presented at: Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing. HPCVirt '09. Nuremberg, Germany; 2009:17-24; ACM, New York, NY.
9. Shi L, Chen H, Sun J, Li K. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans Comput*. 2012;61(6):804-816. https://doi.org/10.1109/TC.2011.112.
10. Duato J, Peña AJ, Silla F, Mayo R, Quintana-Ortí ES. rCUDA: reducing the number of GPU-based accelerators in high performance clusters. Paper presented at: Proceedings of the 2010 International Conference on High Performance Computing Simulation. Caen, France; 2010:224-231.

11. Diab KM, Rafique MM, Hefeeda M. Dynamic sharing of GPUs in cloud systems. Paper presented at: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum. IPDPSW '13; 2013:947-954; IEEE Computer Society, Washington, DC.
12. Zhao X, Zhang Y, Su B. Multitask oriented GPU resource sharing and virtualization in cloud environment. In: Wang G, Zomaya A, Martinez G, Li K, eds. *Algorithms and Architectures for Parallel Processing*. Cham: Springer International Publishing; 2015:509-524.
13. Nvidia *Nvidia Multi-Process Service. vR418*. Santa Clara, CA, U.S.: Nvidia Corporation; 2019.
14. Jiménez VJ, Vilanova L, Gelado I, Gil M, Fursin G, Navarro N. Predictive runtime code scheduling for heterogeneous architectures. In: Seznec A, Emer J, O'Boyle M, Martonosi M, Ungerer T, eds. *High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg/Germany: Springer; 2009:19-33.
15. Choi HJ, Son DO, Kang SG, Kim JM, Lee H-H, Kim CH. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *J Supercomput*. 2013;65(2):886-902. https://doi.org/10.1007/s11227-013-0870-6.
16. Gregg C, Boyer M, Hazelwood K, Skadron K. Dynamic heterogeneous scheduling decisions using historical runtime data. Paper presented at: Proceedings of the Workshop on Applications for Multi-and Many-Core Processors (A4MMC). San Jose, USA; 2011.
17. Wen Y, Wang Z, O'Boyle MFP. Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. Paper presented at: Proceedings of the 2014 21st International Conference on High Performance Computing (HiPC). Goa, India; 2014:1-10.
18. Shulga DA, Kapustin AA, Kozlov AA, Kozyrev AA, Rovnyagin MM. The scheduling based on machine learning for heterogeneous CPU/GPU systems. Paper presented at: Proceedings of the 2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW). St. Petersburg, Russia; 2016:345-348.
19. Rafique MM, Cadambi S, Rao K, Butt AR, Chakradhar S. Symphony: a scheduler for client-server applications on coprocessor-based heterogeneous clusters. Paper presented at: Proceedings of the 2011 IEEE International Conference on Cluster Computing. Austin, TX, USA; 2011:353-362.
20. Trejo-Sánchez JA, López-Martínez JL, García JOG, Ramírez-Pacheco JC, Fajardo-Delgado D. A multi-agent architecture for scheduling of high performance services in a GPU cluster. *Int J Combinat Optim Probl Inform*. 2018;9(1):12-22.
21. Rodriguez MA, Buyya R. Container-based cluster orchestration systems: a taxonomy and future directions. *Softw Pract Exp*. 2019;49(5):698-719. https://doi.org/10.1002/spe.2660.
22. Joseph CT, Chandrasekaran K. Straddling the crevasse: a review of microservice software architecture foundations and recent advancements. *Softw Pract Exp*. 2019;49(10):1448-1484. https://doi.org/10.1002/spe.2729.
23. Civolani Lorenzo, Pierre Guillaume, Bellavista Paolo. FogDocker: start container now, fetch image later. Paper presented at: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing. Auckland New Zealand; 2019:51-59.
24. Medel Víctor, Rana Omer, Bañares José ángel, Arronategui Unai. Modelling performance & resource management in kubernetes. Paper presented at: Proceedings of the 9th International Conference on Utility and Cloud Computing. UCC'16; 2016:257-262; New York, NY, ACM.
25. Medel V, Tolón C, Arronategui U, Tolosana-Calasanz R, Bañares JÁ, Rana OF. Client-side scheduling based on application characterization on kubernetes. In: Pham C, Altmann J, Bañares JÁ, eds. *Economics of Grids, Clouds, Systems, and Services*. Cham: Springer International Publishing; 2017:162-176.
26. Cérin C, Menouer T, Saad W, Abdallah WB. A New Docker Swarm Scheduling Strategy. 2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2), Kanazawa. Kanazawa, Japan; 2017:112-117. https://doi.org/10.1109/SC2.2017.24.

How to cite this article: El Haj Ahmed G, Gil-Castiñeira F, Costa-Montenegro E. KubCG: A dynamic Kubernetes scheduler for heterogeneous clusters. *Softw Pract Exper*. 2020;1–22. https://doi.org/10.1002/spe.2898