Resource Management for TensorFlow Inference

Luciano Baresi, Giovanni Quattrocchi(✉), and Nicholas Rasi

Dipartimento di Elettronica, Informazione e Bioingegneria,
Politecnico di Milano, Milan, Italy
{luciano.baresi,giovanni.quattrocchi,nicholas.rasi}@polimi.it

Abstract. TensorFlow, a popular machine learning (ML) platform, allows users to transparently exploit both GPUs and CPUs to run their applications. Since GPUs are optimized for compute-intensive workloads (e.g., matrix calculus), they help boost executions, but introduce resource heterogeneity. TensorFlow neither provides efficient heterogeneous resource management nor allows for the enforcement of user-defined constraints on the execution time. Most of the works address these issues in the context of creating models on existing data sets (*training* phase), and only focus on scheduling algorithms. This paper focuses on the *inference* phase, that is, on the application of created models to predict the outcome on new data interactively, and presents a comprehensive resource management solution called *ROMA (Resource Constrained ML Applications). ROMA* is an extension of TensorFlow that (a) provides means to easily deploy multiple TensorFlow models in containers using Kubernetes b) allows users to set constraints on response times, (c) schedules the execution of requests on GPUs and CPUs using heuristics, and (d) dynamically refines the CPU core allocation by exploiting control theory. The assessment conducted on four real-world benchmark applications compares *ROMA* against four different systems and demonstrates a significant reduction (>75%) in constraint violations and 24% saved resources on average.

1 Introduction

TensorFlow [1] is one of the most used machine learning (ML) framework in industry [10] and shares similar functionality with other solutions such as PyTorch [19] or MXNet [5]. While TensorFlow supports different types of ML applications, this paper focuses on *supervised learning* ones because of the two phases that characterize their lifecycle: *training* and *inference*. In the former case, algorithms like logistic regression, decision trees, and deep neural networks are used to create prediction models starting from known input-output pairs

This work has been partially supported by the SISMA national research project, which has been funded by the MIUR under the PRIN 2017 program (Contract 201752ENYB) and by the European Commission grant no. 825480 (H2020), SODALITE.

H. Hacid et al. (Eds.): ICSOC 2021, LNCS 13121, pp. 238–253, 2021.
https://doi.org/10.1007/978-3-030-91431-8_15

(e.g., pictures and contained objects), called training set. In the latter case, generated prediction models are used as oracles to infer the result on new, unknown inputs. The first phase makes these applications batch ones, while the second phase requires that these applications be *interactive*.

Both phases are characterized by highly parallel operations (e.g., matrix calculus) that can exploit multi-core architectures. TensorFlow eases the use of multi-core CPUs and also of GPUs, which provide hundreds of cores and very fast executions. Oftentimes, these applications are executed in the cloud, where virtual machines (VMs) equipped with GPUs and dedicated execution frameworks can easily be rented from many cloud providers.

TensorFlow (similarly to other ML frameworks) does not allow users to define constraints on response times (Service Level Agreements or SLA) for these applications, and resource management is driven by user experience or by simple default policies that do not take actual application needs into account. Training would call for deadlines, that is, constraints on the maximum span of batch processing [21], while inference calls for average response times, computed on a number of subsequent invocations over a predefined time window.

Several approaches in the literature focus on the resource management of ML training [3,12], while the inference phase calls for new studies and approaches. Existing solutions applied to interactive web applications [2,7] cannot be reused since they do not consider the heterogeneity introduced by GPUs but only different types of virtual machines. CPUs and GPUs are interdependent resources while different VMs are not. GPUs are faster than CPUs but they also use CPUs to load and write data, and to be activated. Moreover, they have different scaling capabilities: CPUs can precisely be scaled by allocating fractions of cores to single applications; GPUs can only be time-shared among applications. While faster GPUs alone are usually not enough to serve realistic workloads, the coordinated use of CPUs and GPUSs becomes mandatory to offer reasonable execution times.

On the other hand, solutions that combine the management of CPUs and GPUs target the training phase (or long-lasting processing), they focus on scheduling and loadbalancing algorithms, and do not consider dynamic resource provisioning [17,18]. Finally, in inference mode the distributed heterogeneous execution of multiple concurrent ML applications is still not completely supported in TensorFlow (as in other similar tools) and users are required to manually configure their deployments.

This paper presents *ROMA*, an extension of TensorFlow that helps the deployment and oversees the inference phases of multiple concurrent ML applications deployed onto a shared cluster of nodes that offer both CPUs and GPUs. *ROMA* manages containerized TensorFlow models, automates their deployment using Kubernetes[1], a well known container orchestrator, and allows users to define SLAs as constraint on the response time. *ROMA* enacts the control at three different levels. A centralized component exploits heuristics to prioritize the scheduling of application requests on GPUs or CPUs according to their needs.

[1] https://kubernetes.io.

Distributed control-theoretical planners allocate the amount of CPUs needed to each application by considering the boost introduced by GPUs. An intermediate level handles resource contentions that could happen when the system saturates.

The evaluation based on four real-world applications shows that *ROMA*: i) enables the distributed concurrent execution of multiple applications on heterogeneous resources ii) minimizes the number of SLA violations (reduction >75%) compared to static and rule-based solutions, and a simplified control-theoretic approach, and iii) optimizes the use of cluster resources by avoiding unneeded allocations (24% resource saving on average).

The rest of the paper is organized as follows. Section 2 introduces *ROMA*, its architecture and deployment model. Section 3 presents how the schedulers work, and Sect. 4 explains the employed control-theoretical planners. Section 5 shows the empirical evaluation we carried out to assess *ROMA*. Section 6 discusses the related work and Sect. 7 concludes the paper.

2 *ROMA*

ROMA[2] is a comprehensive resource management solution that eases the deployment and operations of multiple interactive ML applications. *ROMA* can be useful to both users interested in running their ML applications and service providers. In the former case, *ROMA* helps the user manage resources efficiently and meet set response times. In the latter case, *ROMA* allows the service provider to allocate fewer resources to each application and offer an higher level solution to users (ML as-a-service).

ROMA is an extension of TensorFlow but it can be easily integrated onto other ML platforms. TensorFlow, as other similar frameworks, does not provide any dedicated support to distribute the inference of new results on computed trained models, neither it takes into account concurrent executions specifically. An extension, called *TensorFlow Serving*[3] (*TF Serving* for brevity) permits users to expose a trained model by means of a built-in web server and a dedicated REST API but the distributed deployment is not supported. *ROMA* wraps TF Serving instances into containers using *Docker*[4]. Docker also provides means to allocate and share CPU cores among multiple processes through CPU quotas. GPUs can be mounted on Docker containers by using external tools, as the NVIDIA Container Toolkit[5].

The deployment of TF Serving containers is enacted using *Kubernetes*. Kubernetes manages *Pods*, that are, groups of co-located containers and volumes, which bind ephemeral containers to persistent data stores. *Deployments* manage the deployment of pods, along with the number of needed replicas, and how they can be upgraded and configured. *Services* bring communication among related pods by adding shared networking, load-balancing, and external access.

[2] Source code is available at https://github.com/deib-polimi/ROMA.

[3] https://www.tensorflow.org/tfx/guide/serving.

[4] https://www.docker.com.

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

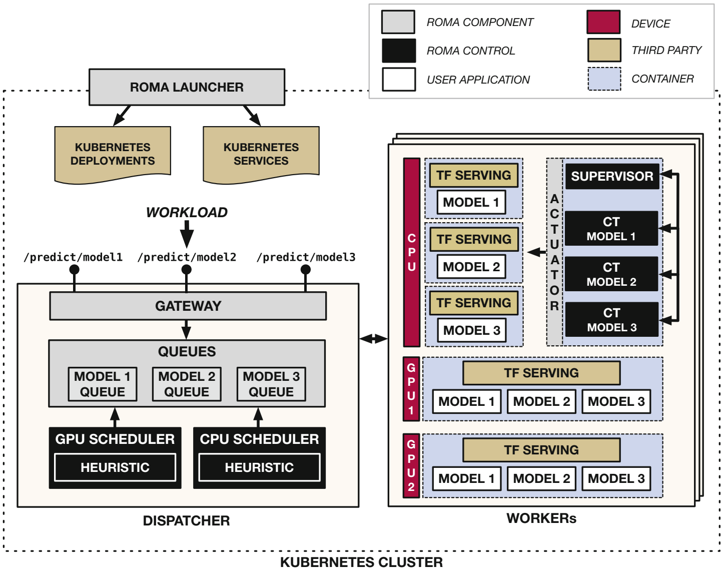


Fig. 1. *ROMA*.

Kubernetes also offers dedicated plugins for AMD and NVIDIA boards (the NVIDIA Container Toolkit is then required) to exploit GPUs [15], but a single GPU cannot be associated with more than one container, and fractions of GPUs cannot be requested (they can only be allocated as complete units).

2.1 Architecture

Figure 1 shows the architecture of *ROMA* while managing three ML models. *ROMA* uses a centralized node, called *dispatcher*, and multiple distributed nodes, called *workers*. *Dispatcher* allows users to add trained models (applications), receives inference (execution) requests, and uses schedulers to distribute these executions on *workers*' *devices*. Each *worker* provides one or more *devices*, that is, at least one CPU and zero or more GPUs.

ROMA deploys *model executables*, that are containers wrapping a TF Serving instance loaded with one or more models, as Kubernetes pods into *workers*. For each managed model, multiple *model executables* (i.e., replicas) can be deployed onto different *workers* to handle intense workloads. Each *model executable* can be instructed to process a request on CPUs or GPUs. Moreover, *model executables* are deployed onto *workers* along with a dedicated control theory-based controller (*CT Controller)* in charge of the fine-grained allocation of CPU cores.

Gateway accommodates requests in dedicated execution queues, one for each application (i.e., trained model). Requests are kept in the queues waiting for execution, that is, waiting for a GPU or CPU to become available. Requests are removed from the queues and assigned for execution to *model executables* by two different schedulers, one for GPUs and one for CPUs. The two schedulers exploit different heuristics to prioritize requests and instruct *model executables* to process them on either a GPU or a CPU.

GPU Scheduler extracts requests from the queue of the model with the greatest difference between expected and measured performance (see Sect. 3) to boost executions. *CPU Scheduler* works together with *CT Controllers*. It removes requests from queues by using a fair round-robin policy and instructs the proper *model executables* to use CPU cores to process them. *CT Controllers* accelerate or decelerate these executions by continuously modifying the CPU cores allocated to *model executables*. Their control period is extremely fast (i.e., 1 s) and allocated resources are changed on the fly, without restarting *model executables* (vertical scalability).

When *GPU Scheduler* instructs a *model executable* to process a request by using a GPU, the average time needed to execute that *model executable* abruptly decreases[6]. Distributed *CT Controller*s handle this sudden change and react by decreasing the number of allocated CPU cores. Note that allocated cores could not be lowered even when GPUs operate because of other external factors (e.g., workload fluctuations).

Given that multiple *CT Controllers* work on the same *worker* node, their combined resource demand can be greater than the actual capacity of the node: a *Supervisor* deployed onto each *worker* oversees demands and manage contentions. Collected data on resource demand, contention, and execution times can then be used to deploy new *model executables* and new *worker*s, but this is out of the scope of this paper. Both schedulers and supervisors exploit lightweight heuristics to be reactive and manage incoming requests properly.

In the case of extremely high workloads, the *dispatcher* can easily be replicated to accommodate a higher level of parallelism without any changes to the underlying control strategies. In this case, clients connect either directly to one of the available replicas or to an additional load balancer that in turn distributes the traffic to the *dispatcher*s. Then, each *dispatcher* can work independently of the others by only scheduling the traffic portion it receives. Local *CT Controllers* just need to be informed of the amount of requests executed by the GPUs without any additional knowledge on the deployment of the other components. *Workers* can be managed by a single designated *dispatcher* or shared among multiple ones. In the latter case, the multiple schedulers would not interfere with one another since their algorithms only use application-level performance data that are locally measured by each *dispatcher*.

2.2 Deployment

As soon as a user submits a trained model, along with its SLA, *ROMA Launcher* generates or updates required Kubernetes *deployments* and *services* to let the system deploy and manage the *model executables*.

[6] This average execution time is computed by considering the different executions of the same *model executable* over a given time window.

ROMA uses two strategies to deploy *model executables*. The user can set the number of to-be-deployed replicas for each model. Replicas can also be added and removed dynamically according to application needs. The placement of *model executables* can either balance their number on the different nodes or deploy them onto the same *worker* until a predefined number of replicas is reached. Note that *ROMA* does not allow one to deploy multiple replicas of the same model on the same *worker* node for the same device. If *model executables* need more resources on the fly, *CT Controllers* takes care of it without creating new replicas.

To exploit the different *devices*, each *model executable* is bounded to a specific *device.* In particular, given m models selected to be deployed onto a worker node, *ROMA* provisions: (i) m *model executable* containing one model each, and binds them to the node's CPU(s), (ii) one *model executable*, containing all models, for each GPU, and (iii) one container that includes the *CT Controllers* of all models, the *Supervisor*, and one actuator implemented as a Kubernetes volume. This means that since we assume that the worker depicted in Fig. 1 comes with two GPUs, and it manages three models, *ROMA* deploys six containers in total.

This deployment allows *ROMA* to exploit the means provided by Kubernetes for using GPUs on each model and also to exploit the CPUs when needed. As already said, the *Supervisor* and models' *CT Controllers* manage CPU cores. As for CPUs *ROMA* deploys a different container for each model because resources can be allocated to them independently. Since GPUs cannot be shared among multiple containers, nor can their cores be allocated to different models, a single container per GPU with all models is enough. The *GPU Scheduler* is in charge of electing the model that can exploit the GPU to serve the next inference request (this is done by calling an internal, model-specific TensorFlow Serving endpoint). At each control step, *ROMA* uses an actuator based on *Docker out of Docker (DooD)* to provide on-the-fly reconfiguration of running containers. DooD is a volume that provides means to launch Docker commands (e.g., to re-configure a container) within another container[7].

3 Schedulers and Supervisors

The goal of *ROMA* is to fulfill constraint over the response time. While in the following we constrain the *average* response time, more conservative metrics (e.g., high percentiles) would only require a stricter set-point and more used resources, and would provide additional tail-latency guarantees. However, our evaluation (see Sect. 5) shows that even by only constraining the *average*

[7] In December 2020, the Kubernetes team announced that the Docker runtime will be considered deprecated in future versions [14]. Docker will not be removed from Kubernetes at least until late 2021. While the evaluation of *ROMA* in Sect. 5 is based on the described Docker-dependent implementation, we are already developing a version of *ROMA* that does not require Docker and that supports other container runtimes as, for example, *containerd* [6].

response time, *ROMA* provides a lower maximum response time than other competitor approaches (e.g., rule-based).

Given a model m, the average response time computed over a given time window w can be formulated as follows.

$$\tau_{R_m} = \frac{\sum_{g=1}^{G}(\tau_{Q_g} + \tau_{P_g}) + \sum_{c=1}^{C}(\tau_{Q c} + \tau_{P_c})}{G + C} \tag{1}$$

Algorithm 1. GPU Scheduling

```
function FREEGPU($gpu$)
    $\mathcal{E} = []$
    $\mathcal{M} \leftarrow getModels()$
    for $m \in \mathcal{M}$ do
        $q \leftarrow m.getQueue()$
        $\mathcal{T}_E \leftarrow []$
        for $n \leftarrow 0, n < q.length, n{+}{+}$ do
            $req \leftarrow q[n]$
            $\tau_Q = now() - req.getTimeIn()$
            $\mathcal{T}_E.append(\tau_Q + n * \tau_{PG_m})$
        end for
        $comp \leftarrow m.getCompletedRequests()$
        $\mathcal{T}_R \leftarrow []$
        for $req \in comp$ do
            $\mathcal{T}_R.append(req.getRT())$
        end for
        $\tau_{R_m} \leftarrow avg(\mathcal{T}_R)$
        $\tau_{E_m} \leftarrow avg(\mathcal{T}_E)$
        $\tau_{W_m} = \beta * \tau_{E_m} + (1 - \beta) * \tau_{R_m}$
        $\tau^{\circ}_{R_m} \leftarrow \alpha * \tau_{SLA_m}$
        if $\tau_{W_m} \leq \tau^{\circ}_{R_m}$ then
            $\epsilon_m \leftarrow 0$
        else
            $\epsilon_m \leftarrow (\tau_{W_m} - \tau^{\circ}_{R_m})/\tau^{\circ}_{R_m}$
        end if
        $\mathcal{E}.append(\epsilon_m)$
    end for
    $m_S \leftarrow \mathcal{M}[\mathcal{E}.indexOf(max(\mathcal{E}))]$
    $req \leftarrow m_S.getQueue().pop()$
    $gpu.execute(req)$
end function
```

Algorithm 2. Supervisor

```
$cs = getControllers()$
$\mathcal{U}_C \leftarrow []$
for $c \in cs$ do
    $u_C \leftarrow c.nextAllocation()$
    $\mathcal{U}_C.append(u_C)$
end for
$AC \leftarrow MC - GC$
$\eta \leftarrow AC/sum(\mathcal{U}_C)$
for $c \leftarrow 0, c < cs.length, c{+}{+}$ do
    $u_C \leftarrow \mathcal{U}_C[c]$
    if $\eta \leq 1$ then
        $u'_C \leftarrow u_C * \eta$
    else if $\eta > 1$ then
        $u'_C \leftarrow (1 - \gamma) * u_C * +\eta * \gamma * u_C$
    end if
    $cs[c].updateStateAndActuate(u'_C)$
end for
```

where G and C are the numbers of requests executed on the GPUs and CPUs respectively in w, τ_{Q_i} is the time spent by a request i in the queue, while τ_{P_i} is the time spent by a GPU or a CPU to process request i. An SLA on τ_{R_m} can state that:

$$\tau_{R_m} <= \alpha \cdot \tau_{SLA_m} = \tau^{\circ}_{R_m} \tag{2}$$

where τ_{SLA_m} is the threshold on the response time defined in the SLA for model m and α is a parameter, which ranges between 0 and 1, that defines the set point $\tau^{\circ}_{R_m}$ for model m. If $\alpha = 1$ then the set point matches τ_{SLA_m}; lower values are more conservative and let the system tolerate more imprecision.

As already said, *ROMA* distributes the processing of requests to the different *devices* in the cluster by means of the two dedicated schedulers. Their goal is to select both which request to execute next and on which *device*. The rationale is that *GPU Scheduler* always selects the request of the model with the "highest" needs (see below). To complement GPUs, requests are also scheduled for processing onto CPUs by means of a round-robin policy where the (non-empty) queues to serve are selected randomly. Note that *CPU Scheduler* could find queues empty if the GPUs are fast enough to process all the workload alone.

GPU Scheduler is activated in an event-based fashion. Function *freeGPU* (Algorithm 1) is executed as soon as a GPU (parameter *gpu*) becomes free, that is, at system startup and when a GPU completes the execution of a request. In particular, we designed a heuristic that, for each model m, takes into account a weighted average (τ_{W_m}) of measured response times (τ_{R_m}) and of the estimated response times of the requests that are in the queues waiting to be processed (τ_{E_m}). The estimation is computed by using the accumulated queue time of each request (τ_Q in Eq. 1) and the profiled processing time on GPUs τ_{PG_m}. Parameter β, which ranges in interval $[0, 1]$, defines the weight associated with τ_{R_m} and τ_{E_m}. A higher value of β gives more importance to requests in the queue and makes the system more responsive to workload bursts. Given the computed averaged response time τ_{W_m}, the distance from the set point $\tau^{\circ}_{R_m}$ is computed as ϵ_m (lines 21–25). The selected model m_S is the one with the highest ϵ_m. The first request in queue m_S is the one that is processed by *gpu* using the proper *model executable* (lines 28–30).

The actual allocation of CPU cores is managed by the *CT Controllers* associated with the different *model executables*. For this reason, *CPU Scheduler* dispatches requests to CPU *device*s using a round robin policy. *CPU Scheduler* repeatedly removes a request from a randomly selected queue and schedules it for CPU execution on a randomly selected *model executable*. This way the load sent by *CPU Scheduler* to each *model executable* is homogeneous and the burden of managing CPU allocation is handled locally by *CT Controllers*. Each *worker* is associated with a *Supervisor* in charge of refining the resource allocation computed by *CT Controllers* in case of contention. At each control step (1 s), a *CT Controller* computes the amount of CPU cores u_C (core allocation demand) needed by its *model executable*, which embeds model m, to meet set response time $\tau^{\circ}_{R_m}$ (as described in Sect. 4). Each *CT Controller* computes its u_C independently of the others, that is, they do not communicate.

Supervisors use the heuristic shown in Algorithm 2 to compute a feasible core allocation u'_C for each *CT Controller* deployed on a *worker*. First, all the core allocation demands u_C are gathered in a vector $\mathcal{U}_C$ (line 1–6). Being MC the total number of cores provided by the *worker*, and GC the number of CPU cores statically allocated to support GPU execution, the difference between MC and GC is the actual amount of cores that can be allocated (AC) to *model executables* (line 7) in a given *worker*. As mentioned before, GPUs and CPUs are interdependent since the former consume the processing power of the latter to load data in memory and to be activated. Note that if GC is set to 0, GPUs will slow down requests running on CPUs. This is seen by *CT Controller* as another disturbance that is naturally mitigated by the control logic (described in Sect. 4). Moreover, η is the ratio between AC and the sum of all demanded cores, that is, the sum of all u_C (line 8). Given η, each u'_C is computed as follows. If η is less than 1, the actual demand cannot be fulfilled since demanded cores are more that available ones (under provisioning). Each u'_C is then computed by multiplying each u_C by η (line 12). If η is equal to 1, the amount of demanded cores matches available ones (AC), and $u'_C = u_C$. If η is greater than 1, available cores are over provisioned. However, we introduce parameter γ to maximize resource utilization (line 14). The default value ($\gamma = 0$) implies that $u'_C = u_C$. If γ is between 0 and 1, we allocate more cores and obtain more responsive models. $\gamma = 1$ means that all AC cores are always used. Finally, the state of each *CT Controller* is updated using u'_C and computed core allocation is actuated.

4 Controllers

To design the *CT Controller* we need a dynamic model[8] for the relationship between the CPU and GPU allocation (u_C, u_G) and the response time τ_R; u_C and u_G jointly modify the output rate r_o from the queue, the input rate r_i being an exogenous disturbance. *CT Controllers* do not require any knowledge of the application structure (i.e., of the operations to execute on input data) and the same dynamic model is general enough to support different kinds of compute- and GPU-intensive interactive applications (e.g., machine learning inference, scientific calculus, graph-based computations), with proper profiling. This is possible because the proposed controllers are grey-box, that is, their model does not include all aspects of the system but just the ones that describe its physics. The employed fast feedback-loop (control period equals to 1 s) is in charge of correcting the imperfections of the model at runtime. Here we represent the compound of the above in a simplified manner (yet adequate, as the reported tests will show) as an additive perturbation, and we set:

$$\begin{cases} \tau_R(t) = \tau_Q(t) + \tau_P(t) \\ \tau_Q(t) = \frac{\ell(t-\tau_Q(t))}{r_o(t)} \\ \frac{d\ell(t)}{dt} = r_i(t) - r_o(t) \\ r_o(t) = r_{on}\big(u_C(t), u_G(t)\big) + d_o(t) \\ \tau_P(t) = \tau_{Pn}\big(u_C(t), u_G(t)\big) + d_P(t) \end{cases} \tag{3}$$

where τ_Q the time spent on the queue, τ_P is the processing time downstream of the queue depending on (u_P, u_G) through a nominal relationship $\tau_{Pn}(\cdot,\cdot)$ with an additive disturbance d_P, and $r_{on}(\cdot,\cdot)$ is the $(u_C, u_G) \rightarrow r_o$ relationship in some "nominal" condition, and $d_o(t)$ the combined effect of all the disturbances.

Model (3) explains the physics of the system, but is not suitable *as is* for control design owing to the contextual presence of a differential equation and an implicit one with delay. It however evidences that under the above assumptions, response time control boils down to queue length control. From Eq. (3) one notices that (i) at steady state r_o has to balance r_i but this can happen for any ℓ, hence (ii) a steady-state variation of τ_R is obtained by *transiently* causing an

[8] To avoid ambiguities, in this section a *dynamic model* is a mathematical representation of the controlled system, that is, of the ML application.

input/output rate imbalance via u_C and/or u_G, and then restoring the balance once the desired τ_R is achieved as the new queue length, divided – mind the balance – by the through rate, gives the necessary τ_Q, hence τ_R.

$$\begin{cases} \frac{d\ell(t)}{dt} = r_i(t) - r_o(t) \\ r_o(t) = \mu_C u_c(t) + \mu_G u_G(t) \\ \tau_R(t) = \frac{\ell(t)}{r_o(t)} \end{cases} \tag{4}$$

where the gains μ_C and μ_G account for the processing speed of CPUs and GPUs, respectively, and the delay is considered negligible with respect to the control time scale. Linearised in the vicinity of an operating point described by nominal values of the throughput and the required waiting time, $\overline{r}_o$ and $\overline{\tau}_R$ to name them,

Overall, therefore, the compound of the above gives rise to the continuous-time transfer function description:

$$\Delta\tau_R(s) = G_{\tau_R C}(s)\Delta u_C(s) + G_{\tau_R G}(s)\Delta u_G(s) \tag{5}$$

where uppercase letters denote the Laplace transform of the corresponding lowercase variables and:

$$G_{\tau_R C}(s) = -\frac{\mu_C}{\overline{r}_o}\,\frac{1+s\overline{\tau}_R}{s}, \quad G_{\tau_R G}(s) = -\frac{\mu_G}{\overline{r}_o}\,\frac{1+s\overline{\tau}_R}{s} \tag{6}$$

where s is the Laplace transform complex variable. Transforming (6) to discrete time, we conclude that a physically grounded $\mathscr{Z}$-transform model (denoting by z the corresponding complex variable, i.e., the one-step advance operator) takes the form:

$$\Delta\tau_R(z) = G^*_{\tau_R C}(z)\Delta u_C(z) + G^*_{\tau_R G}(z)\Delta u_G(z) \tag{7}$$

where

$$G^*_{\tau_R C}(z) = -\,k_C\,\frac{z-b}{z-1}, \quad G^*_{\tau_R G}(z) = -\,k_G\,\frac{z-b}{z-1} \tag{8}$$

Parameters k_C, k_G, and b can be obtained online by profiling the applications of interest and fitting measured responses to those of the dynamic model. In this work we assume that when a GPU takes part of the work —which is represented as a step-like behaviour of u_G— the CPU attempts to restore the required τ_R so as to free the GPU as soon as possible. This means requiring that the closed-loop transfer function from u_G to τ_R has a zero in $z = 1$. The said transfer function then becomes:

$$F_o(z) = \frac{z-1}{z-p} \tag{9}$$

where parameter $p \in [0, 1]$ governs the required response speed: $p \to 0$ means faster response, $p \to 1$ slower. This gives controller

$$G_c(z) = \frac{(k_C - 1)z^2 + (2 - k_C p - k_C b)z + k_G bp - 1}{k_C(z-1)(z-b)} \tag{10}$$

i.e., a real PID. To further reduce computational complexity, we however decided to employ a PI controller, that is,

$$G_c(z) = K\frac{z-a}{z-1} \tag{11}$$

and prescribe the closed-loop poles to coincide in $z = q$, where q is interpreted as p above. This is achieved by setting:

$$K = \frac{4(a-1)(b-1)}{k(b-a)^2}, \quad a = \frac{(2-b)q - b}{q - 2b + 1} \tag{12}$$

while the presence of integral action ensures zero steady-state errors.

5 Evaluation

This section describes the experiments we carried out to evaluate the feasibility and benefits of *ROMA*.

To run the experiments, we deployed *ROMA* on a cluster of three virtual machines on Microsoft Azure: one VM of type $HB60rs$ with a CPU with 60 cores and 240 GB of memory for the *dispatcher*, and two VMs, as *worker* nodes, of type $NV6$ equipped with a NVIDIA Tesla $M60$ GPU and a CPU with 6 cores and 56 GB of memory. We also used an additional instance of type $HB60rs$ for generating the client workload.

The experiments exploited four existing ML applications: *Skyline Extraction* [9], *ResNet* [11], *GoogLeNet* [20], and *VGG16* [22]. The first application uses a combination of computer-vision algorithms to extrapolate the horizon skyline from a set of images and the others perform classification tasks. In particular, *ResNet* exploits a residual neural network, while *GoogLeNet* (*G.Net*) and *VGG16* employ two different deep convolutional neural networks. All these four models were trained and then used in inference mode with companion sample images.

ROMA ($\alpha = 0.8$, $\beta = 0.5$ and $\gamma = 0$) was set to use a static deployment strategy and we deployed all applications onto the two *worker* nodes. We statically reserved $GC = 0$ cores for the GPUs, to say that the additional disturbances introduced by the usage of CPUs for loading and operating GPUs are handled by *CT Controllers*. These controllers were manually tuned: $K = 0.15$ and $a = 0.11$.

We compared *ROMA* against the four exemplar systems we implemented by using a different heuristic for the *GPU Scheduler* and/or another type of controllers instead of *CT Controllers*. All these systems used a round robin scheduler (*RR*) for GPUs. In addition, system *RR+rules* used a rule-based controller that allocated 1 additional CPU core to a *model executable* if the response time is greater than or equal to $0.8 * SLA$. If the response time is equal to or less than

Table 1. SLA and workloads.

Test	Apps	SLA	Workload
2-Apps	*Skyline*	0.38	20-20-80-80-20-20-20-20
	G.Net	0.45	20-20-20-20-70-70-20-20
2-Apps	*ResNet*	0.54	40-30-20-40-30-20-30-30
	VGG16	0.56	20-30-40-20-40-40-30-30
AllApps	*Skyline*	0.38	10-10-10-10-30-30-30-10
	G.Net	0.45	10-30-30-30-10-10-10-10
	ResNet	0.54	10-20-30-10-20-30-10-20
	VGG16	0.56	10-30-20-10-30-20-10-20

Table 2. Comparison.

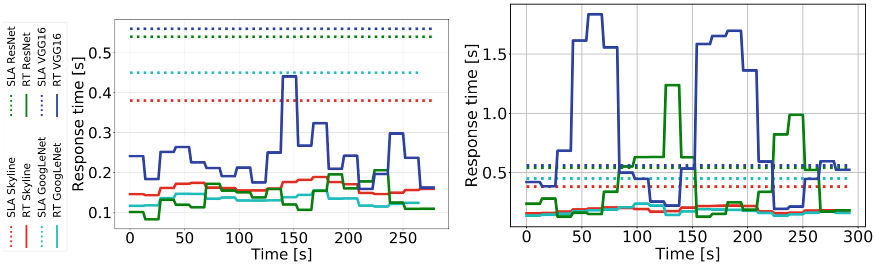
Test	System	τ_R	τ_{R_M}	τ_{R_σ}	V	Res
2-Apps *G.Net* *Skyline*	*ROMA*	0.168	0.245	0.031	0	748
	RR+CT	0.193	0.358	0.053	0	793
	RR+rules	0.185	0.638	0.075	10	1165
	RR+max	0.152	0.191	0.014	0	3600
	RR+min	0.278	1.053	0.182	40	600
2-Apps *ResNet* *VGG16*	*ROMA*	0.266	0.553	0.068	10	1633
	RR+CT	0.446	0.949	0.117	70	1750
	RR+rules	0.701	3.829	0.528	120	1691
	RR+max	0.337	0.711	0.092	40	3600
	RR+min	1.537	4.423	0.464	180	600
AllApps	*ROMA*	0.167	0.427	0.022	0	1767
	RR+CT	0.325	1.372	0.130	90	2018
	RR+rules	0.409	1.913	0.158	140	1973
	RR+max	0.208	0.453	0.052	0	3600
	RR+min	1.032	6.828	0.414	170	600

$0.2 * SLA$ it de-allocated a core. The control period was set to 15 s. System *RR+CT* used the same *CT Controllers* as *ROMA* for managing CPU resources. The control period was set to 1 s. System *RR+max* statically allocated all cores (6 per *worker*) fairly distributed to applications. System *RR+min* statically allocated a minimum amount of cores (1 per *worker*) equally distributed to applications.

We tested the systems by running two concurrent applications at a time (test *2-Apps*): i) *GoogLeNet* and *Skyline Extraction* and ii) *ResNet* and *VGG16*. We repeated each test 3 times for a total of 60 executions (5 systems, 4 applications, and 3 executions). Table 1 shows the SLAs (in seconds) and workloads (in incoming requests per second) used in the experiments. Each experiment lasted 300 s and the workload of each application was changed with a different step (shown in column $Workload$) every 37 s (8 times). Table 2 shows the *average* (τ_R) and *maximum* response times (τ_{R_M}) in seconds along with the standard deviation (τ_{R_σ}), the number of SLA violations (V), and the number of allocated CPU resources (Res) measured as $cores * seconds$.

With the first application pair, *ROMA* produced 0 violations and a resource allocation equal to 748 (where the lower means the better). *RR+rules* allocated 1.5 times the resources used by *ROMA* without avoiding SLA violations and obtained longer response times. *RR+CT* performed similarly to *ROMA*, but *ROMA* allocated GPUs in a smarter way (i.e., lower average and maximum response times) and thus relying on CPUs less frequently, which means saving a greater amount of resources. The allocation of all cores makes *RR+max* the fastest system, but by using more than 5 times the CPU resources utilized by *ROMA*. Finally, *RR+min* consumed fewer resources than the other systems at the cost of obtaining 40 SLA violations.

With the second application pair, *ROMA* obtained 10 SLA violations and a resource allocation of 1633 $cores * seconds$. Once again *ROMA* was able to



(a) *ROMA* (b) *RR+rules*

Fig. 2. System experiments - *All Apps*.

outperform the other systems showing a better balance between violations and resource usage, and lower average and maximum response times. Given the presence of *VGG16*, the use of GPUs was fundamental to make the system serve the incoming workload. Results show that a round robin scheduling of GPUs was not sufficient to avoid SLA violations even if all the CPU cores were always allocated statically (*RR+max* produced 40 violations). Compared to *ROMA*, *RR+rules* showed a higher response time and 120 SLA violations and an allocation of almost the same amount of CPU resources. Even with a smarter allocation of CPUs (*RR+CT*) the obtained response time was almost double the one measured with *ROMA* and the number of SLA violations were 70. *RR+min* violated the *SLAs* 180 times and also presented an average response time greater than 1.5 s (almost three times greater than set SLAs).

As final experiment, we ran the four applications concurrently (test *All Apps*) for a total of 60 additional executions (4 applications, 5 systems, 3 repetitions each). Table 2 presents obtained results and the charts of Fig. 2 show the response times obtained with *ROMA* and with *RR+rules* (the best competitor) using the workloads and SLAs reported in Table 1. *ROMA* was able to always keep the response time under the SLAs (0 violations), with an overall average response time equals to 0.167 s, a maximum response time of 0.427 s, and allocated 1767 $cores * seconds$. In contrast, *RR+rules* frequently violated the SLAs while executing *VGG16* and *ResNet*, and resulted in slower executions (average and maximum response times equal to 0.409 and 1.913 s, respectively). *RR+CT* obtained 90 violations and higher response times than *ROMA*, while *RR+max* obtained 0 violations but allocated 3600 $cores * seconds$. The combined use of the heuristic that favors executions on GPUs for resource-hungry applications and its control theory-based CPU allocation made *ROMA* not only faster but also able to exploit fewer resources than all the other systems (except w.r.t. *RR+min* that violated the SLA 170 times).

6 Related Work

Several solutions deal with the management of heterogeneous resources at the node level but not GPUs. For example, the solution presented by Lakew et al. [16] exploits control theory to provision multiple resources dynamically to satisfy SLAs. Similarly to *ROMA*, they exploit containers and can reconfigure resources dynamically. Farokhi et al. [8] present a fuzzy control approach that coordinates the autoscaling of CPU cores and memory. They show that the coordinated control of multiple resources outperforms the performance of the same system with independent controllers.

These approaches manage complementary resources: CPUs uses memory (and also disks) for completing a task, while *ROMA* exploits competing resources since a request can be executed on either CPUs or GPUs. This means that *ROMA* must consider both scheduling and resource provisioning while aforementioned works focus only on the latter.

Different approaches focus on the management on GPUs and CPUs. For example, Khadil et al. [13] present OSched, a resource-aware scheduler for OpenCL jobs that aims to maximize the throughput of the hosting infrastructure. Chen et al. [4] propose a solution for improving the performance of MapReduce applications by scheduling map and reduce tasks on CPUs and GPUs using heuristics. They compared their approach with CPU-only and GPU-only versions of the system obtaining an improvement between 20% and 110%.

Compared to these works, *ROMA* is different from both the control and application domain point of views. First, the mentioned approaches focus on the scheduling of computing tasks on GPUs and CPUs, while *ROMA* combines both scheduling and fine-grained resource allocation in a comprehensive solution. *ROMA*'s scheduling heuristics cooperate with control-theoretical planners in order to minimize constraint violations while optimizing resource usage. Second, existing solutions focus on the management of GPUs in the context of long-lasting compute intensive applications (e.g., machine learning training jobs), while *ROMA* focus on interactive ML applications. To the best of our knowledge *ROMA* is the first solution that provides an architecture, a deployment model and a comprehensive resource management approach for ML inference.

7 Conclusions and Future Work

The paper presents *ROMA*, an extension of TensorFlow that eases the management and operation of ML applications executed on a cluster of heterogeneous resources (GPUs and CPUs) in inference mode. *ROMA* allows users to constrain applications execution times and exploits scheduling heuristics and control-theory based resource provision to run them efficiently. The assessment of the work uses four real-world applications and shows promising results.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th Symposium on Operating Systems Design and Implementation, pp. 265–283. USENIX (2016)
2. Baresi, L., Guinea, S., Leva, A., Quattrocchi, G.: A discrete-time feedback controller for containerized cloud applications. In: Proceedings of the 2016 24th International Symposium on Foundations of Software Engineering, pp. 217–228. ACM (2016)
3. Baresi, L., Leva, A., Quattrocchi, G.: Fine-grained dynamic resource allocation for big-data applications. IEEE Trans. Softw. Eng. **47**(8), 1668–1682 (2021)
4. Chen, L., Huo, X., Agrawal, G.: Accelerating MapReduce on a coupled CPU-GPU architecture. In: Hollingsworth, J.K. (ed.) SC Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11. IEEE/ACM (2012)
5. Chen, T., Li, M., et al.: MXNet: a Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv (2015)
6. Containerd: an industry-standard container runtime with an emphasis on simplicity, robustness and portability (2021). https://containerd.io
7. Ding, J., Cao, R., Saravanan, I., Morris, N., Stewart, C.: Characterizing service level objectives for cloud services: realities and myths. In: 2019 IEEE International Conference on Autonomic Computing (ICAC), pp. 200–206. IEEE (2019)
8. Farokhi, S., Lakew, E.B., Klein, C., Brandic, I., Elmroth, E.: Coordinating CPU and memory elasticity controllers to meet service response time constraints. In: 2015 International Conference on Cloud and Autonomic Computing, pp. 69–80 (2015)
9. Fedorov, R., Camerada, A., et al.: Estimating snow cover from publicly available images. IEEE Trans. Multimed. **18**(6), 1187–1200 (2016)
10. Forbes: TensorFlow Turns 5 - Five Reasons Why it is the Most Popular ML Framework. http://tiny.cc/Forbes-TF (2020)
11. He, K., Zhang, X., et al.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
12. Jahani, A., Lattuada, M., Ciavotta, M., Ardagna, D., Amaldi, E., Zhang, L.: Optimizing on-demand GPUs in the cloud for deep learning applications training. In: 2019 4th International Conference on Computing, Communications and Security (ICCCS), pp. 1–8 (2019)
13. Khalid, Y.N., Aleem, M., Prodan, R., Iqbal, M.A., Islam, M.A.: E-OSched: a load balancing scheduler for heterogeneous multicores. J. Supercomput. **74**(10), 5399–5431 (2018)
14. Kubernetes: Don't Panic: Kubernetes and Docker (2020). https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker
15. Kubernetes: Schedule GPUs (2020). https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/
16. Lakew, E., Papadopoulos, A., Maggio, M., Klein, C., Elmroth, E.: KPI-agnostic control for fine-grained vertical elasticity. In: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 589–598. IEEE (2017)
17. Mittal, S., Vetter, J.S.: A survey of CPU-GPU heterogeneous computing techniques. ACM Comput. Surv. **47**(4), 69:1–69:35 (2015)

18. Nozal, R., Bosque, J.L., Beivide, R.: EngineCL: usability and performance in heterogeneous computing. Future Gener. Comput. Syst. **107**, 522–537 (2020)
19. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Proc. Syst. **32**, 8024–8035 (2019)
20. Szegedy, C., et al.: Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–9 (2015)
21. Verma, A., Cherkasova, L., et al.: Deadline-based workload management for MapReduce environments: pieces of the performance puzzle. In: 2012 IEEE Network Operations and Management Symposium, pp. 900–905. IEEE (2012)
22. Zhang, X., Zou, J., He, K., Sun, J.: Accelerating very deep convolutional networks for classification and detection. IEEE Trans. Pattern Anal. Mach. Intell. **38**(10), 1943–1955 (2015)