2024 IEEE/ACIS 22nd International Conference on Software Engineering Research, Management and Applications (SERA) | 979-8-3503-9134-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SERA61261.2024.10685647

Leveraging Shared Accelerators in Kubernetes Clusters with rOpenCL

Rui Alves
Instituto Politécnico de Bragança
Campus de Santa Apolónia
5300-253 Bragança, Portugal
rui.alves@ipb.pt,
ORCID 0000-0003-4128-8779

José Rufino
Research Centre in Digitalization and Intelligent Robotics (CeDRI)
Laboratório para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC)
Instituto Politécnico de Bragança, Campus de Santa Apolónia
5300-253 Bragança, Portugal
rufino@ipb.pt, ORCID 0000-0002-1344-8264

***Abstract*—High-Performance Computing involves exploiting the capabilities of powerful computing resources to achieve high computational performance, efficiency, and scalability. By enabling to process complex tasks and/or large amounts of data efficiently, HPC plays an essential role in multiple domains. Parallelization and distribution of workloads across multiple processing resources is at the core of HPC and has been done via many approaches, including different programmatic models and service frameworks. In recent years, HPC has been coupled with Containerization, as a way to deploy, manage and run workloads in an automated and efficient manner, including those relying on accelerators. The performance of heterogeneous workloads, however, depends on the behavior of the platform scheduler, and of the mechanisms used to share accelerators, being also constraint by the limited number of accelerators that each node may host. This paper describes an approach by which the last limitation may be overcome, using a Kubernetes cluster as a show case. The approach builds on the integration of Kubernetes with rOpenCL, an OpenCL API forwarder for remote execution of OpenCL calls. This combination may also be particularly useful in Edge Computing scenarios, where containerized edge services gain the ability to access, transparently and efficiently, remote (e.g. cloud-based) accelerators. Some preliminary experimental results are presented, demonstrating the feasibility of this integration, and the impact on performance for an OpenCL application.**

***Index Terms*—Heterogeneous Computing, Containerization, OpenCL, Kubernetes**

I. Introduction

Crucial in scientific research, engineering, and many industries and services, where large-scale computational capabilities are essential, High-Performance Computing [1] refers to the use of advanced computing techniques and systems to solve complex problems that require significant computational power and/or the ability to deal with very large datasets. To this extent, specialized architectures and combinations of resources are often required, to cope with the increasing demands for computational power, ubiquitous computing, and power (energy) consumption/optimization concerns.

These requisites meet together, for instance, in approaches such as Edge Intelligence [2], which is becoming important in Internet of Things (IoT) contexts, and expect processing and analysis of data to occur near the data source, typically at the "edge" of the network, rather than relying solely on centralized cloud servers. Edge Intelligence thus benefits from leveraging the variety of computing resources that may be local or near, and which often involve some form of specialized accelerators, like GPUs, FPGAs, or other co-processors [3]. Thus, heterogeneous computing plays an important role in this context, exposing a mix of different processors that may be used to optimize the performance of specific workloads.

Edge Intelligence also benefits from its combination with container management platforms, like Kubernetes (K8s). Such integration [4] enables the orchestration of diverse edge services, providing a robust framework for deploying intelligent applications at the edge while addressing the challenges associated with resource constraints and network limitations on edge devices. This combination enhances Edge Intelligence solutions' overall agility, scalability, and reliability in decentralized computing scenarios.

The importance of both Heterogenous Computing and Containerization for Edge Intelligence makes the case for their integration within that context. Such, however, carries with it several challenges [5]: integrating heterogeneous systems, each with its architectural specificity and programming model, within a K8s environment, can be complex, requiring careful consideration of compatibility and abstraction layers; the K8s API provides a unified orchestration layer, but the varied capabilities and performance characteristics of the different accelerators make it challenging to ensure both a fair and efficient utilization of those co-processors. Moreover, the number of accelerators available in each node, is necessarily limited, thus restraining the number of hosted containers that depend on accelerators. Furthermore, migrating containers to different nodes, in order to exploit faster accelerators that may have become free, requires check-pointing mechanism that are not pre-programmed in most heterogeneous applications.

In this paper, an alternative approach to the management of accelerators within a K8s cluster is proposed, whereby the definition of the accelerators to be used in each OpenCL workload is no longer determined by the native mechanisms of K8s. Instead, by integrating rOpenCL (remote OpenCL)

This work was supported by national funds through FCT/MCTES (PIDDAC): CeDRI, UIDB/05757/2020 (DOI: 10.54499/UIDB/05757/2020) and UIDP/05757/2020 (DOI: 10.54499/UIDP/05757/2020); and SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P/0007/2020).

with K8s, OpenCL applications running within K8s containers are able to use, and share, without restrictions, both local and remote accelerators, which translate in performance gains, and helps to maximize the utilization of the cluster accelerator set.

The rest of the paper is organized as follows: section II is an overview of the problem; section III presents the proposed solution; section IV provides the results of a preliminary evaluation of the approach proposed; section V concludes the paper and defines some directions for future work.

II. Background

Kubernetes (K8s) [6] is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications, including ones that depend on co-processors. This possibility is particularly valuable, as denoted by the fact that many organizations have been increasingly deploying and exploiting K8s clusters that incorporate such kind of computational resources [7].

Despite the significant improvements made by the K8s community in supporting heterogeneous computing, there are certain limitations and challenges [8] intrinsic to the realization and management of such model within a K8s cluster.

One notable limitation, which is particularly relevant for workloads that require co-processors, lies in the fact that K8s doesn't have a native scheduler specifically designed for accelerator management. The scheduler is responsible for deciding where to run pods (set of $\geq$1 containers) based on the available resources and constraints within the cluster. However, when pods need accelerators, the scheduler isn't able to make smart decisions because it doesn't know the requirements of the workloads and the characteristics of the various accelerators available in the different cluster nodes.

Moreover, contention for the same or overlapping resources is solved by serializing access to them: if two workloads request the same accelerator, or there are more workload requests for accelerators than accelerators available in the K8s cluster pool, the workloads that fail to get the number of accelerators required will be put on hold (pending state) until the resources needed become available. This may lead to performance degradation or instability of the pods.

To minimize resource contention, and its consequences, mechanisms like *Node Affinity* and *Pod Prioritization* may be of help. *Node Affinity* allows to constrain which nodes are eligible to be scheduled based on node labels, which helps to influence the placement of pods on specific nodes in the cluster. This feature allows assigning GPUs (or other accelerators) to label-based pods. However, for some execution contexts, it may be a complex process to select a specific device to use [9]. *Pod Prioritization* is a feature designed to prioritize high-priority pods over lower-priority ones. However, for GPU workloads, applying this may not guarantee a more efficient use of GPU resources than FCFS scheduling, once high-priority pods do not necessarily maximize GPU usage. Moreover, depending on the compute time needed by the workloads running within such pods, they may lead to the under-utilization of GPUs for a considerable amount of time.

Some solutions to overcome these limitations originated on GPU vendors. NVIDIA, for instance, supports an over-subscription strategy, allowing to deploy multiple workloads in the same accelerator, by leveraging the GPU's time-slicing scheduler [10]. This mechanism for improving GPU utilization may be applied in the K8s environment and is transparent to applications. When this mechanism is configured, a certain number of GPU replicas is defined, so that the GPU is time-shared by as much pods (each pod may only be assigned one GPU replica). However, because the number of replicas is static, when the number of pods requiring the GPU exceeds the number of replicas, the pods in excess will remain in the pending state until a replica becomes available for them. This situation is illustrated in Fig. 1, where two replicas of the same GPU are available; thus, the green pods were successfully assigned a GPU replica, while the red pod awaits for one.

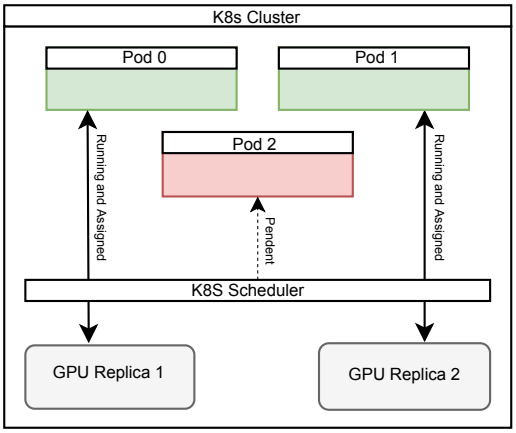

Fig. 1. A K8s cluster with Pods requiring Co-Processors.

III. Coupling Kubernetes with rOpenCL

This section describes the integration of rOpenCL [11], [12] with K8s to alleviate the contention previously described.

rOpenCL was developed to allow OpenCL applications to exploit remote accelerators exposed by networked hosts. Its main characteristics are *communication portability* (the network layer is TCP-based and programmed on BSD sockets) and *full transparency* (supports pre-compiled binary OpenCL applications). It relies on two basic components: a *driver*, fully integrated with the ICD-Loader mechanism, installed on the hosts where OpenCL applications may start; a *service*, installed in each remote node whose co-processors are meant to be exposed to the OpenCL applications (for details see [11]).

The approach here proposed builds on the scalability and flexibility of rOpenCL. The scalability derives from the fact that the rOpenCL driver is thread-safe, and both this driver and the remote services are multi-threaded (Pthreads-based). In the driver, a distinct thread is used per each TCP connection with a remote service, which is kept open for the duration of the OpenCL application that originated it. In the services, the corresponding TCP endpoints are managed by specific threads, which receive the OpenCL calls forwarded by the clients,

979-8-3503-9134-3/24/$31.00 ©2024 IEEE

SERA 2024, May 30-June 1, 2024, Honolulu, USA

submit them to the local OpenCL platforms, and return the results. At any moment, service threads may be mediating OpenCL requests from many different OpenCL applications, whose host component may be running at any networked node where the rOpenCL driver was installed. Thus, an OpenCL application with checkpoint capabilities (e.g., Hashcat), or a service with an OpenCL backend, may be stopped and restarted anywhere, as long as the rOpenCL driver is present, which allows to see the same global set of co-processors; when repeating the OpenCL bootstrap (platform and devices identification), it is then possible to keep using the same set of co-processors, or change it considering their locality (co-processors that were previously local may now be remote, and vice-versa, and rOpenCL exposes the IP address of their platform as an attribute), or even the results of on-the-fly micro-benchmarks conducted to tune the workload balancing.

Fig. 2 is a representation of the scenario that emerges by joining the concepts of K8s cluster and rOpenCL cluster. In this context, a rOpenCL cluster is a set of rOpenCL nodes, and these are any systems (physical or virtual) running the rOpenCL service to mediate access to local co-processors for which there are OpenCL drivers (platforms) installed.

The K8s and rOpenCL clusters may be implemented by a disjoint set of hosting nodes (physical or virtual), or they may overlap. Thus, an rOpenCL node may be also hosting K8s pods, or these may be hosted in separated nodes. As long as there is TCP/IP connectivity between them, and pods have the rOpenCL driver pre-installed, their interoperability is ensured.

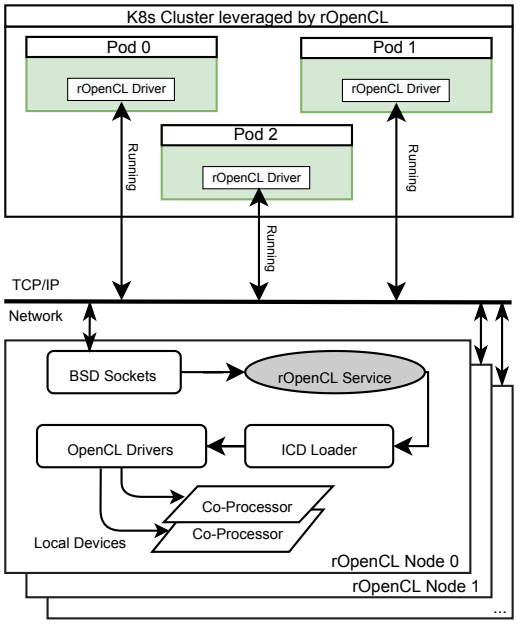

Fig. 2. A K8s cluster leveraged by rOpenCL.

In a K8s-only scenario, when a pod requires GPU resources, its YAML configuration includes the number of GPUs required (e.g. *nvidia.com/gpu: 1*). The K8s scheduler then uses this information during the pod scheduling phase, considering the node labels, and ensuring that the pod is placed on a node with the appropriate GPU resources. When K8s is coupled with rOpenCL, OpenCL workloads deployed in a K8s cluster no longer need any GPU requirement to be stated in the YAML configuration of their pods. Thus, the K8s scheduler now manages them as "normal" pods / workloads, completely unaware of their dependency on those types of resources, as access to the accelerators is now provided collaterally via rOpenCL. This means that the scheduler can now put these workloads on any node of the K8s cluster, without having to worry about the management of their co-processor requirements, and, most importantly, not enforcing the sequential use of those resources by the pods, or having to manage the use of GPU replicas.

In addition, the new approach allows workloads to always have access to the particular combination of co-processors they find suitable, regardless of the migrations they may suffer triggered by the K8s scheduler. On the other hand, OpenCL workloads may avoid possible negative impacts of migration by exploiting the K8s *Node Affinity* mechanism: a pod may find the particular combination of local and remote devices it uses to be optimal and, in order to avoid any performance deterioration coming from the change of the co-processors locality, it may decide to stick to the same hosting node for its entire lifetime. The same mechanism may also be used to limit the usable K8s nodes to the ones that overlap with rOpenCL nodes, meaning any pods deployed on those nodes will always have at least one local co-processor available, which may be used preferably over other remotely accessible devices.

IV. Experimental Validation

This section presents the results of a preliminary validation of the integration between K8s and rOpenCL just presented.

The test-bed used was based on 3 virtual machines (VM0, VM1, VM2), running in distinct (but homogeneous) nodes of a KVM-based cluster, served by a 100 Gbps Ethernet network. Each VM was assigned 16 SMT CPU-cores of an AMD EPYC 7452 CPU, and 32 GB of RAM. VM1 and VM2 were assigned (passthrough) one NVIDIA RTX 2080 Ti GPU each. All VMs ran Linux Ubuntu 22.04 64-bit with kernel 5.15. The NVIDIA driver version in the VMs with GPUs was 525.147.05.

The K8s cluster was formed with 2 of the 3 VMs: the VM without any GPU assigned (VM0), assuming the role of control plane; and one of the VMs with a GPU assigned (VM1), used to host the pods created for the tests and where the benchmark application was launched. The rOpenCL cluster was based on VM1 and VM2 (the VMs with GPUs), and thus the K8s and rOpenCL cluster shared one node (VM1).

In VM1, the NVIDIA solution for time-slicing was installed, with the number of GPU replicas set to 4. This limit was defined after concluding that the specific GPU used could support up to 4 simultaneous instances of the test workload.

The hashcat OpenCL application [13] was selected as the test workload. This is a well-known tool within the cybersecurity community, used to test the robustness of passwords and also as a password recovery tool. It has also been used in previous related work [11], once it is able to take advantage of more than one GPU (thus being adequate to exploit remote

GPUs in addition to local ones), and is mostly compute-bound (thus minimizing the impact of network communications).

In this work, hashcat was used to perform a brute-force attack on 10 000 SHA-256 hashes (with a 16 bit random salt), corresponding to the top 10 000 passwords of the file `10-million-password-list-top-10000.txt` from the repository https://github.com/danielmiessler/SecLists, using the wordlist file `rockyou.txt` from the repository https://github.com/brannondorsey/naive-hashcat.

The NVIDIA GPU time-slicing mechanism was first evaluated, on VM1, with 8 different hashcat workloads, each with a successively increasing number of deployed pods: workload 1 with 1 pod, workload 2 with 2 pods, and so on, up to workload 8 with 8 pods. Once the number of GPU replicas was set to 4, this means that for workloads 1 to 4 all pods will be running (time-sharing the GPU), while for workloads 5 to 8 there will be pods pending. For each workload, the overall execution time was measured: the difference between the moment the 1st pod starts, and the moment the last pod finishes.

This procedure was then repeated with the workloads using rOpenCL, in the following way: for workloads 1 to 4, the GPU of the VM1 was shared by the pods involved, using the local rOpenCL driver to redirect the OpenCL requests to the local rOpenCL service; this allows to compare the NVIDIA time-slicing mechanism with rOpenCL, where both allow the sharing of the underlying GPU, but impose different overhead; then, for workloads 5 to 8, the pods in excess of 4 use the rOpenCL driver in VM1 to redirect the OpenCL request to the rOpenCL service in VM2, thus exploiting a remote GPU; this demonstrates the full integration of rOpenCl in the K8s environment and the performance gains it may bring with it.

Figure 3 shows the results of the evaluation conducted. The effect of the remote GPU exposed by rOpenCL is clearly visible, with speedups of 1.13 to 1.84 when the number of pods grows past the number of GPU replicas (4) used in time-slicing. A deceleration introduced by rOpenCL over time-slicing when using the local GPU is also shown (0.78 to 0.9), and is expected, due to the overhead of the network transactions (despite happening within VM1). The figure also allows to compare the total execution time of the workloads.

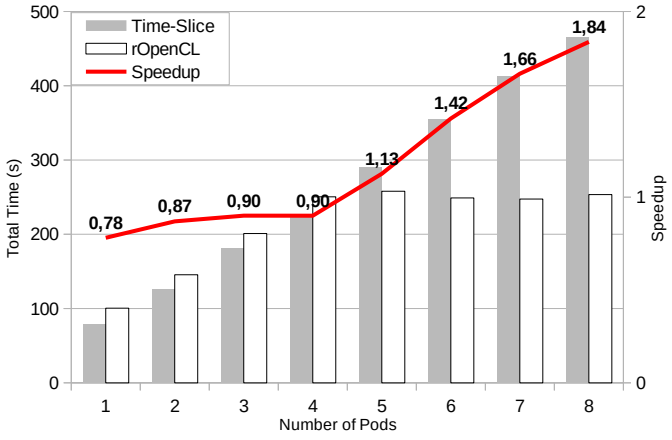

Fig. 3. Comparing time-slicing vs rOpenCL in a K8s scenario.

V. Conclusion

The integration of rOpenCL in K8s, achieved through a pre-installed driver in the pods, frees up K8s from the management of GPU access and sharing by the pods, allowing workloads to use those co-processors in a fully transparent and unrestricted way. Moreover, through rOpenCL, the workloads gain access to a wider set of GPUs, once the full set from the rOpenCL cluster is exposed, and not only the GPUs attached to the node where the pod is hosted. This way, workloads may compose their accelerator set using different criteria, considering the GPU's intrinsic capabilities and the performance tradeoffs originating from their locality (local GPUs vs remote GPUs).

This architecture is also compatible with the migration of pods, as per the decisions of the K8s scheduler, making possible to the pods to keep using the same accelerator set, or change it, as found appropriate. The lack of native support in K8s for live migration of pods also means that the solution proposed is only suitable to move OpenCL workloads that may be stopped and started over in another node, where the OpenCL bootstrap would be repeated; this would be the case, for instance, of a web service with an OpenCL backend to perform some heavy processing on a per request basis.

Despite the advantages presented, the unrestricted sharing of GPUs by the pods, when using rOpenCL, opens the door for device overloading, calling for a future throttling mechanism.

References

[1] D. Reed, D. Gannon, and J. Dongarra, "Reinventing high performance computing: Challenges and opportunities," 2022. [Online]. Available: https://arxiv.org/abs/2203.02544

[2] J. Mendez, K. Bierzynski, M. P. Cuéllar, and D. P. Morales, "Edge intelligence: Concepts, architectures, applications, and future directions," *ACM Trans. on Embedded Computing Systems*, vol. 21, no. 5, oct 2022.

[3] C. P. Filho, E. Marques, V. Chang, L. dos Santos, F. Bernardini, P. F. Pires, L. Ochi, and F. C. Delicato, "A systematic literature review on distr. machine learning in edge computing," *Sensors*, vol. 22, no. 7, 2022.

[4] Q.-M. Nguyen, L.-A. Phan, and T. Kim, "Load-balancing of kubernetes-based edge computing infrastructure using resource adaptive proxy," *Sensors*, vol. 22, p. 2869, 04 2022.

[5] N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe, "Container orchestration on hpc systems through kubernetes," *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 10, no. 1, feb 2021.

[6] C. N. C. Foundation. (2023) Kubernetes API Documentation. [Online]. Available: https://kubernetes.io/docs/reference/kubernetes-api/

[7] D. Meng. (2018) Kubernetes distributed deep learning on heterogeneous gpu clusters. [Online]. Available: https://thenewstack.io/kubernetes-distributed-deep-learning-on-heterogeneous-gpu-clusters/

[8] C. Carrión, "Kubernetes scheduling: Taxonomy, ongoing issues and challenges," *ACM Computing Surveys*, vol. 55, no. 7, dec 2022.

[9] Kubernetes Contributors. Scheduling gpus. [Online]. Available: https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/

[10] NVIDIA. (2023) Nvidia device plugin for kubernetes documentation. [Online]. Available: https://docs.nvidia.com/datacenter/cloud-native/index.html#kubernetes-and-nvidia-gpus

[11] R. Alves and J. Rufino, "Extending heterogeneous applications to remote co-processors with ropencl," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 305–312.

[12] R. Alves and J. Rufino, "Remote execution of opencl and sycl applications via ropencl," in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2023, pp. 51–60.

[13] hashcat Project. (2024) hashcat-advanced password recovery. [Online]. Available: https://hashcat.net/hashcat/