Abstractions for Containerized Machine Learning Workloads in the Cloud

Balaji Subramaniam, Niklas Nielsen, Connor Doyle, Ajay Deshpande, Jason Knight, Scott Leishman
Intel Corporation
balaji.subramaniam@intel.com

1 INTRODUCTION AND MOTIVATION

Many institutions rely on Machine Learning (ML) to meet their goals. ML workloads are computationally intensive and as a result there is an investment in accelerators [2, 9] such as ASICs, FPGAs and GPUs to improve their performance. At the same time, these institutions are increasingly adopting cloud infrastructure with containers gaining traction relative to virtual machines [14, 17].

Designing, implementing and maintaining a system to support ML workloads in a cloud infrastructure with containers that takes resource management into account poses significant challenges. However, what data scientists desire is the flexibility to implement new models and research new ML techniques without having to spend a lot of time managing the underlying platforms and infrastructure. Based on our engagements, we find that they want the following features:

1.1 Layered Abstractions

A fundamental tension arises when designing abstractions for data scientists. On one end of the spectrum, they are satisfied to work with high level, general purpose tooling, pipelines, and workflow configuration. But on the other end, such general purpose abstractions can restrict, rather than enhance their workflow. Instead, data scientists want layered abstractions such that that the underlying primitives can be manipulated directly when necessary.

1.2 Flexible Configuration

A data scientist wanting to run an ML task in the cloud using containers needs to package their experiment in one or more images, push that to a registry, create and run the ML task, and then monitor and potentially troubleshoot its execution. Today, this requires an understanding of the inner-workings of the infrastructure and the container orchestration system itself. As such, there is a high bar for entry and opens up a large surface area for mistakes. We strive to make this process easier for data scientists by providing a set of higher-level APIs that use sensible defaults and appropriate hooks for cluster operators to dictate behavior. Based on the specifications provided by a data scientist, the definition of new job types and their use should be seamless.

1.3 Data Management

ML training tasks are characterized by learning from input data. Given the volume that is typically accessed, data scientists separate their code from this data and instead attach it to their jobs in a declarative fashion at runtime using the scheduling system. Some tasks require this data to be duplicated across jobs, as in hyper-parameter sweeps, or partitioned across nodes, as in distributed training. Additionally, these datasets are often curated dynamically via SQL queries, user specified code, or generated as the output of prior machine learning or preprocessing steps. Additional data management requirements include ease of use, resiliency to failures, and performance optimizations. Example optimizations include scheduling ML tasks on compute resources where the data is already available (data affinity) and pre-populating data on the compute resources if required (data caching).

1.4 Performance Enhancements

In order to extract maximal performance out of a given compute resource, optimizations such as CPU isolation and pinning, NUMA awareness and device locality are required [12, 15, 16]. These optimizations are also expected to be performed without much involvement from the end user. While these optimizations have been studied and used in high-performance computing (HPC), they are in nascent stages of adoption and implementation in cloud container orchestrators.

2 BUILDING BLOCKS IN SUPPORT OF DATA SCIENTISTS

To address some of the aforementioned challenges and requirements, we describe our efforts to design, implement and maintain an end-to-end ML system built on top of Kubernetes [10, 13]. Kubernetes uses a declarative state model to drive objects (e.g. `Pods`, `Services`, `Namespaces`) from their `current` state to their `desired` state, and ultimately execute containerized workloads. The current state and the desired state are represented in the `Status` and `Spec` of each object, respectively. To accomplish this, Kubernetes uses a set of high-level abstractions called `Controllers` (e.g. `Deployment`, `DaemonSet`, `Job`) which can detect and act on state differences in basic objects. As an example, a `Pod`'s definition is considered a declaration of its `Spec`, but when first created its current `Status` will not match its `Spec` and so the `Controller` will carry out tasks like downloading the container image to converge towards the desired state.

In rest of this paper, we present our work to support data scientists using the following abstractions:

2.1 Definition of ML Job Types

We use Custom Resource Definitions (CRDs) and custom controllers [5, 7] to allow for the definition of new ML job types in Kubernetes. CRDs enable extending Kubernetes with new objects. With CRDs, the new ML job types can be created and managed like any other native Kubernetes objects. It acts as a base from which layered abstractions can be built for data scientists.

SysML'18, February 2018, Stanford, CA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We have designed and developed a unified framework for the definition and maintenance of new ML job types in Kubernetes. The goal is to provide a mechanism for seamless definition of new ML job types with minimal code changes. Towards this goal, we make the following contributions: unifying custom controller logic, defining new job types and their subresources via templates, and providing abstractions to interact with CRDs and their subresources.

In order to unify the custom controller logic, we have defined a unified state machine that allows for the definition of state transition across various ML job types. A unified state machine enables us to develop a single reconciliation mechanism to drive towards the desired state specified in the CRD of an ML job. The reconciler periodically inspects the desired and the current state of the ML jobs and their associated sub-resources and takes action if required. The reconciler also performs garbage collection when necessary (e.g., delete orphaned Pods originally created by an ML job via CRD).

2.2 Performance Enhancements

As mentioned earlier data scientists want to extract the best performance out of the infrastructure without worrying too much about the details.

In Kubernetes, the CPU manager [3, 4] component was introduced to enable CPU pinning and isolation. Kubernetes can enforce CPU limits using CFS shares. However, better isolation helps performance in some cases where the containerized workload is affected by factors such as cache affinity, device locality and context switches. The CPU manager assigns exclusive CPUs to containers using the cpuset cgroup controller. Exclusive CPUs are only granted for certain resource request parameter combinations. Additional defaulting for resource request fields in job submissions would further shield data scientists from these details while ensuring good performance for sensitive tasks.

Performance optimizations related to NUMA have been well studied and used in the HPC community. This requires coordination between assignment of different resources such as CPU, accelerator devices and network devices. However, in Kubernetes, these resources are handled by disjoint components. Enabling NUMA awareness in Kubernetes requires coordination between multiple components. We are involved in the efforts [8] to enable NUMA awareness in Kubernetes.

Only a few resources are supported natively by Kubernetes. But new accelerators such as FPGAs, ASICs and GPUs are emerging in the market to support growing needs of ML workloads. However, it is not feasible to add support for every accelerator natively in Kubernetes as the mechanism to enable these devices will be vendor and device specific. In Kubernetes, the device plugin component [1, 6] was introduced to allow for the support, maintenance and monitoring of devices without adding vendor and device specific details in the Kubernetes core components.

2.3 Data Management

Distributed storage is a large research area and one filled with many complexities such as fault tolerance, performance optimizations and user interfaces. Our approach to addressing this is to leverage Kubernetes CRDs to enable a flexible interface to the storage system that can expand over time as functionality is added. The flex volume plugin [11] is used as the initial backing store to implement the desired states described by the CRD specification. Users can request storage through this CRD and how those declarations impact their job scheduling and execution.

The specification of the CRD will allow the data scientists to dynamically define the dataset used with their ML tasks and distribute it based on the type of ML task (e.g., hyper-parameter sweep or distributed training). The controller will be responsible for detecting if the data requirements of an ML job is satisfied when the job is scheduled on a compute node and take appropriate actions. These actions include fetching the data if required and providing that data as a volume to the container and optionally labeling the node indicating the presence of that data. The label can be used in the future to provide data affinity. The controller will also perform garbage collection (e.g., evict data to make space), when required.

3 CONCLUSION AND FUTURE WORK

ML is gaining increasing applicability and are among the most important class of workloads. In this paper, we described our efforts to support ML workloads in a containerized environment. Based on our engagement with data scientists, we presented abstractions to address some of the challenges and requirements that data scientists face in working with containerized environments.

Many interesting challenges remain. Accommodating the needs of data scientist at the pace in which the ML community is growing requires flexibility. Any new feature addition requires changes to multiple abstractions. We are actively working on including new abstraction to support data scientists. An example is distributed job scheduling to support multi-node jobs (e.g., distributed training). In HPC, scheduling techniques such as gang scheduling have been studied and extensively used. However, there is a desire to use HPC-like scheduling patterns in cloud-native container orchestration system to schedule and compute distributed ML tasks. In existing systems like Kubernetes, extensions are required to support this requirement.

REFERENCES

[1] 2017. Device Manager Proposal. (2017). https://github.com/kubernetes/community/blob/master/contributors/design-proposals/resource-management/device-plugin.md

[2] 2017. US Coalesces Plans for First Exascale Supercomputer: Aurora in 2021. (Sept. 2017). https://www.hpcwire.com/2017/09/27/us-coalesces-plans-first-exascale-supercomputer-aurora-2021/

[3] 2018. Control CPU Management Policies on the Node. (2018). https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/

[4] 2018. CPU Management Design Proposal. (2018). https://github.com/kubernetes/community/blob/master/contributors/design-proposals/node/cpu-manager.md

[5] 2018. Custom Resources. (2018). https://kubernetes.io/docs/concepts/api-extension/custom-resources/

[6] 2018. Device Plugins. (2018). https://kubernetes.io/docs/concepts/cluster-administration/device-plugins/

[7] 2018. Extend the Kubernetes API with CustomResourceDefinitions. (2018). https://kubernetes.io/docs/tasks/access-kubernetes-api/extend-api-custom-resource-definitions/

[8] 2018. Hardware topology awareness at node level (including NUMA) Issue #49964 kubernetes/kubernetes. (2018). https://github.com/kubernetes/kubernetes/issues/49964

[9] 2018. Intel Invests $1 Billion in the AI Ecosystem to Fuel Adoption and Product Innovation. (2018). https://newsroom.intel.com/editorials/intel-invests-1-billion-ai-ecosystem-fuel-adoption-product-innovation/

[10] 2018. Kubernetes Documentation. (2018). https://kubernetes.io/docs/home/

[11] 2018. Volumes. (2018). https://kubernetes.io/docs/concepts/storage/volumes/

[12] Akhil Arunkumar, Evgeny Bolotin, Benjamin Cho, Ugljesa Milic, Eiman Ebrahimi, Oreste Villa, Aamer Jaleel, Carole-Jean Wu, and David Nellans. 2017. MCM-GPU: Multi-Chip-Module GPUs for Continued Performance Scalability *(ISCA '17)*. ACM, New York, NY, USA, 320–332. https://doi.org/10.1145/3079856.3080231

[13] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2018. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (April 2018), 50–57. https://doi.org/10.1145/2890784

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. 2015. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 171–172. https://doi.org/10.1109/ISPASS.2015.7095802

[15] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters *(USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

[16] Ugljesa Milic, Oreste Villa, Evgeny Bolotin, Akhil Arunkumar, Eiman Ebrahimi, Aamer Jaleel, Alex Ramirez, and David Nellans. 2017. Beyond the Socket: NUMA-aware GPUs *(MICRO-50 '17)*. ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/3123939.3124534

[17] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study *(Middleware '16)*. ACM, New York, NY, USA, 1:1–1:13. https://doi.org/10.1145/2988336.2988337