High-Performance Virtualized Spark Clusters on Kubernetes for Deep Learning

Performance Study - November 18, 2019

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com

Table of Contents

Executive Summary

A virtualized cluster was set up with both Spark Standalone worker nodes and Kubernetes worker nodes running on the same VMware vSphere® virtual machines. The same Spark image classification workload was run on both Spark Standalone and Spark on Kubernetes with very small (~1%) performance differences, demonstrating that Spark users can achieve all the benefits of Kubernetes without sacrificing performance.

Introduction

In recent years, Kubernetes [1] has become a dominant container orchestration and workload management tool. By separating the management of the application and the infrastructure, Kubernetes makes both the application developer and the IT infrastructure administrator more productive. Through very powerful scheduling, isolation, and resource management tools, Kubernetes allows multiple users to share the same computing framework in a secure fashion. Within VMware, the newly announced Project Pacific [2] brings Kubernetes within the vSphere hypervisor itself, providing additional virtual machine management possibilities.

Apache Spark [3] is a popular platform for parallel computation, complementing Hadoop as a Big Data platform. Previous work from the VMware Performance Engineering team has shown that Spark and Hadoop run on vSphere as fast as on bare metal servers (see, for example, Fast Virtualized Hadoop and Spark on All-Flash Disks: Best Practices for Optimizing Virtualized Big Data Applications on VMware vSphere 6.5 [4]). Recent versions of Spark have been modified to work directly with Kubernetes.

To gauge the performance impact of running a Spark cluster on Kubernetes, and to demonstrate the advantages of running Spark on Kubernetes, a Spark Machine Learning workload was run on both a Kubernetes cluster and on a Spark Standalone cluster, both running on the same set of VMs. The results of those tests are described in this paper.

In the next section, Kubernetes is described in more detail, followed by a comparison of the traditional Spark architecture with Spark on Kubernetes. Then the performance tests are detailed, starting with the hardware and software configuration, the Kubernetes configuration used, the workload containerization process, and, finally, the performance results.

Kubernetes

The basic Kubernetes architecture is shown in Figure 1. Container deployments and lifetimes are managed in collections called pods. The Kubernetes Master manages resources and scheduling through an API interface. Cluster state is stored in an etcd key-value store. Both the Master and the etcd store may be made redundant for high availability. Kubernetes worker nodes (simply referred to as "nodes") are managed through the Kubelet agent.

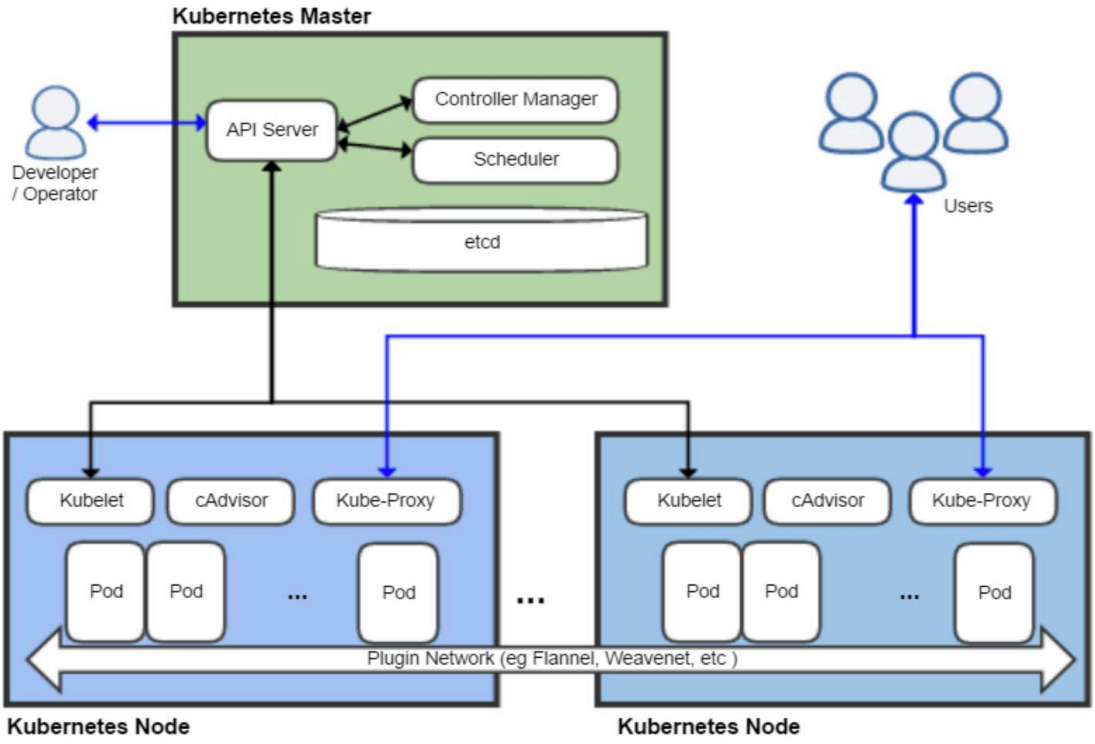

Figure 1: Kubernetes Architecture

Kubernetes operates in both imperative and declarative modes. In the imperative mode, the user issues commands to directly control the state of the cluster through a `kubectl run` or `kubectl create` command, where `kubectl` is the main Kubernetes command-line tool. In the declarative mode, the desired state of the cluster and application is specified in a configuration file. When that configuration file is specified as the argument of a `kubectl apply` command, Kubernetes manages the cluster to implement the desired state, and maintains that state as necessary through the lifecycle of the application, for example, creating new instances as necessary to replace failing replicas.

Running Kubernetes on vSphere brings all the advantages of virtualization including manageability, ease of deployment, and workload consolidation. Additionally, VM-based isolation and multi-tenancy provide additional security and resource management to containers.

VMware currently supports three Kubernetes distributions: Enterprise PKS [5], which is based on Pivotal Container Services and integrated with VMware NSX; Cloud PKS [6] for cloud deployments; and Essential PKS [7], which is closely aligned with open-source Kubernetes. The latter was chosen for this work.

Spark on Kubernetes - Architecture

The standard Spark architecture is shown in Figure 2. Applications are submitted to the Spark driver, which divides up the application into tasks and distributes them to worker nodes running one or more Spark executors, with the cluster manager handling the resources allocated to the workers. For Spark running under Hadoop distributions, the cluster manager typically has been the Hadoop resource manager, YARN (Yet Another Resource Manager).

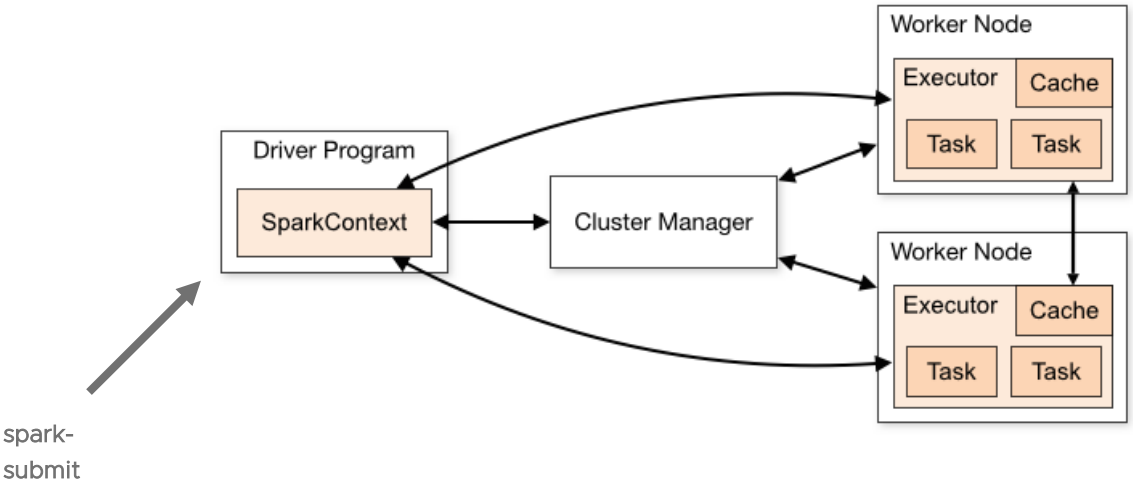

Figure 2: Standard Spark Architecture

For Spark on Kubernetes, the Kubernetes scheduler provides the cluster manager capability as shown in Figure 3. Upon receiving a spark-submit command to start an application, Kubernetes instantiates the requested number of Spark executor pods, each with one or more Spark executors. The Spark driver may run in a pod (Spark Cluster mode) or directly in a VM (Spark Client mode). The latter was used here.

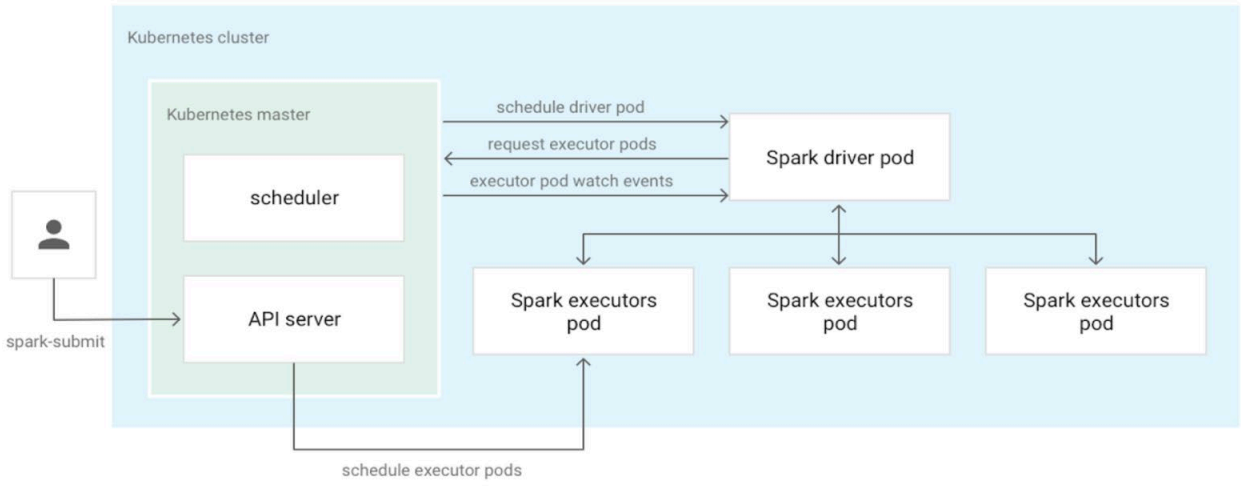

Figure 3: Spark on Kubernetes Architecture

Spark applications running in Standalone mode require that every Spark worker node be installed with the correct version of Spark, Python, Java, etc. This puts a burden on the IT administrator, who may be managing many Spark applications with different requirements, and it requires coordination between the administrator and the application developer. With Kubernetes, the developer only needs to create a container with the correct software, and the IT administrator just needs to manage the cluster using the fine-grained resource management tools to enable the different Spark workloads.

Spark on Kubernetes - Performance Tests

Hardware and Software Configuration

The tests were run on four virtualized (VMware ESXi™ 6.7.3) 2nd-generation Intel® Xeon® Scalable processor ("Cascade Lake") servers, each with 2x Intel Xeon Platinum 8260 CPUs @ 2.4GHz, with a total of 96 logical cores with hyperthreading on, and 768 GB of memory. Five CentOS 7.6 VMs were created on each host, with four beefy worker nodes configured with 16 vCPUs and 120 GB each and a fifth, smaller VM with 8 vCPUs and 64 GB. The total allocation per host was 72 vCPUs and 544 GB.

For Spark Standalone, the smaller VM on Host 1 was set up as the Spark Master (Figure 4). This is where the Spark driver was run. The 16 larger nodes were configured as Spark workers. All Spark nodes were installed with the same versions of Spark, Java, and Python.

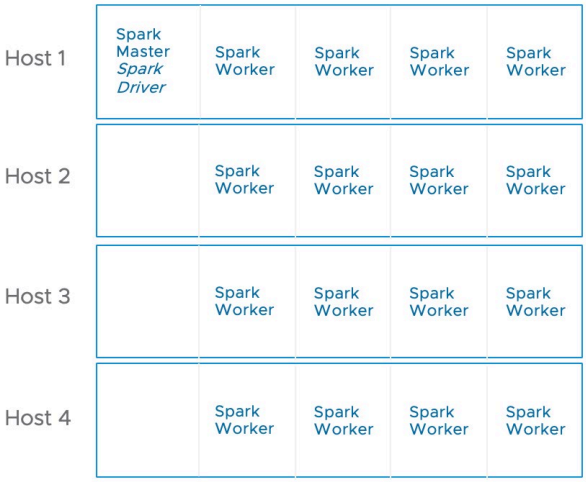

Figure 4: Spark Standalone VM Configuration

To enable the performance comparison between Spark Standalone and Spark on Kubernetes, a VMware Essential PKS cluster was installed on the same VMs (Figure 5). The smaller VM on the first three hosts was set up as a highly available control plane with three stacked masters, fronted by an HAProxy load balancer running on the small VM on the fourth host. Each of the three masters ran a redundant copy of the etcd key-value store containing the cluster configuration data.

The 16 larger VMs served as Kubernetes worker nodes, with each Kubernetes worker node hosting one Spark executor pod, which may contain one or more Spark executors.

The Spark driver ran on the first Kubernetes master. One important difference between this configuration and the Spark Standalone configuration is that, in the Kubernetes cluster, the Spark components only need to be installed in the VM hosting the Spark driver.

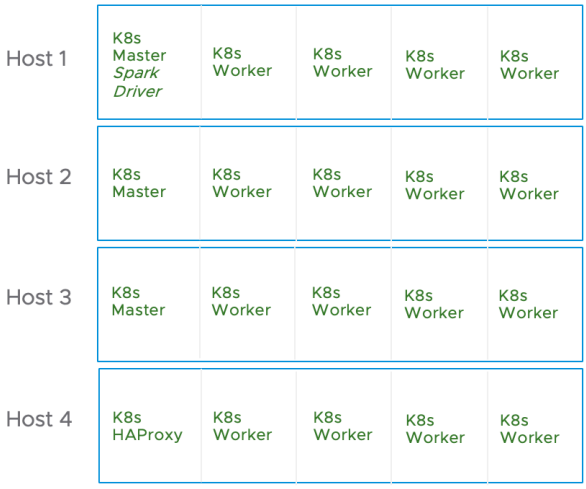

Figure 5: Spark on Kubernetes VM Configuration

Spark Workload Containers

Spark 2.4 ships with sample Docker build files to build containers to run Spark programs in Scala, Python, or R. These were used as a base to create container images for the workloads used in these tests. To customize these containers, the developer can work inside the container instance and then issue a `docker commit` command to create the image when complete. A better method, used here, is to add commands to the Dockerfile to customize it for the desired workloads. The resulting Dockerfile is a self-documenting description of what is in the container. The customized container image was pushed to a local repository available to all nodes using `docker push`.

The workload used in this test classified images using a pre-trained ResNet50 convolutional neural network model. The Spark application was written using Intel BigDL [8] which uses Spark to distribute the neural network operations across Spark executors, calling TensorFlow through Keras.

The dataset used was ImageNet [9], a publicly available set of over one million complex labeled images, with 1,000 classes.

The program used was the Maximum Throughput Spark BigDL ResNet50 image classifier from VMware IoT Analytics Benchmark [10].

For these image classification workloads, Python 3.6, TensorFlow, Keras, and BigDL code were installed in the container. The complete contents of the Dockerfile are shown in "Appendix A: Dockerfile Used to Create Spark Container."

Spark Submit Command

Once the Kubernetes cluster is up and the Spark workload container is prepared, there are two required steps and a recommended step prior to running the Spark application. The recommended step is to create a private namespace for Spark to enable Spark-specific resource quotas and access control policies. For example:

```
kubectl create namespace spark
```

The required steps are to create a Kubernetes service account and give it the ability to create pods:

```
kubectl create serviceaccount spark --namespace=spark

kubectl create clusterrolebinding spark-role --clusterrole=edit \
  --serviceaccount=spark:spark --namespace=spark
```

The Spark application run command, `spark_submit`, has been updated to call on a K8s master rather than a Spark master using the k8s:// prefix. In addition to this change, it is necessary to specify the location of the container image, the service account created above, and the namespace if one was created. Aside from that, the `spark-submit` command is unchanged, as shown in Table 1, with all Kubernetes-specific parameters highlighted in red.

Spark Standalone	Spark on Kubernetes
spark-submit	spark-submit
--master spark://192.168.1.1:7077	--master k8s://192.168.1.2:6443
--driver-memory 40G	--driver-memory 40G
--conf spark.executor.instances=16	--conf spark.executor.instances=16
--conf spark.cores.max=240	--conf spark.cores.max=240
--conf spark.executor.cores=15	--conf spark.executor.cores=15
--executor-memory 100g	--executor-memory 100g
	--conf spark.kubernetes.container.image=192.168.1.1:5000/spark-tf
	--conf spark.kubernetes.namespace=spark
	--conf spark.kubernetes.authenticate.driver.serviceAccountName=spark
--jars <path to jar>	--jars <path to jar>
<program> <arguments>	<program> <arguments>

Table 1: Spark Submit Command - Spark Standalone vs. Spark on Kubernetes

Results

Deep Learning Image Classification on Spark with BigDL

To compare Spark Standalone performance to Spark on Kubernetes performance, the Maximum Throughput Spark BigDL ResNet50 image classifier was run on the same 16 worker nodes, first while configured as Spark worker nodes, then while configured as Kubernetes nodes. Then the number of nodes was reduced by four (by removing the four workers on host 4), and the same comparison was made using 12 nodes, then 8, then 4.

The relative results are shown in Figure 6. The Spark Standalone and Spark on Kubernetes performance in terms of images per second classified was within ~1% of each other for all configurations. Performance scaled well for the Spark tests as the number of VMs increased from 4 (1 server) to 16 (4 servers), although it should be pointed out that the base configuration of 4 VMs was artificially fast since all the VMs were on the same server and thus there was no intra-worker node network traffic. Without this boost, the scaling from 4 to 16 VMs would be closer to the 4x ideal scaling.

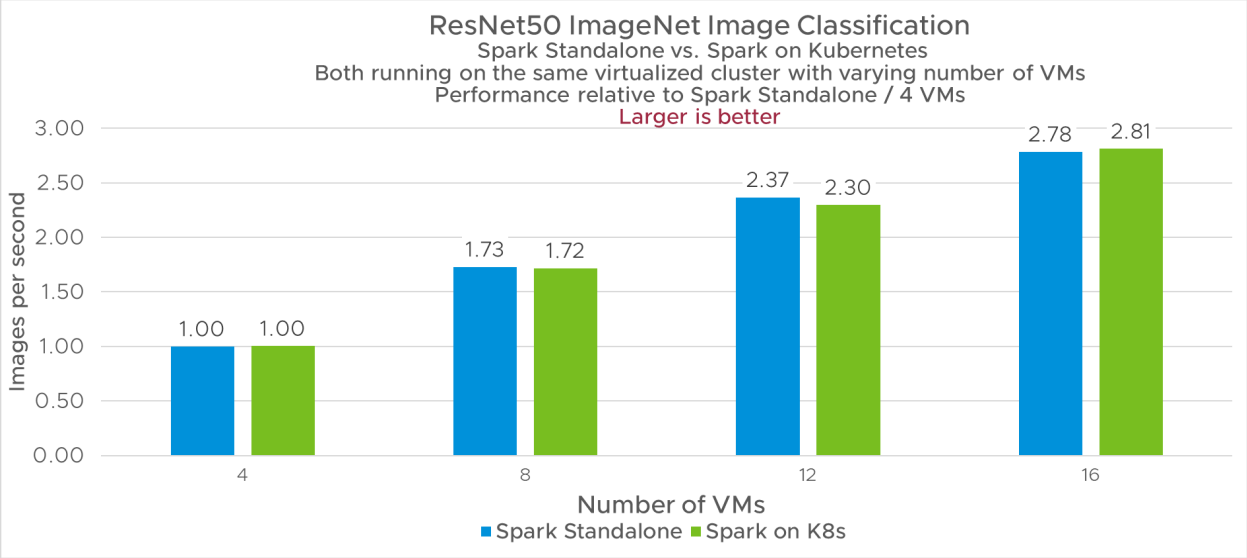

Figure 6: Deep Learning Image Classification on Spark with BigDL

Kubernetes Multiple Workload Performance

A further test was run to determine how well Kubernetes can handle multiple simultaneous workloads running on the cluster. The same Spark Maximum Throughput Spark BigDL ResNet50 image classifier was run on the 16-node Kubernetes cluster, while four image classification tests from the MLPerf Inference benchmark [11] were run on four Kubernetes nodes on one server, using 80% of CPU on that server. The resulting degradation in the Spark test was only 3%, demonstrating the ability of Kubernetes to schedule resources among multiple workloads.

Conclusion

The benefits of running Spark on Kubernetes are many: ease of deployment, resource sharing, simplifying the coordination between developer and cluster administrator, and enhanced security. A standalone Spark cluster on vSphere virtual machines running in the same configuration as a Kubernetes-managed Spark cluster on vSphere virtual machines were compared for performance using a heavy workload, and the difference imposed by Kubernetes was found to be insignificant.

Containers provide a great mechanism for application packaging and Kubernetes provides the management of those deployed containers. When you want to isolate workloads on the same servers from each other, however, then virtual machines provide that strong isolation, as seen in the mixed workloads example.

VMware has made a considerable investment in this key technology and will continue to do so.

Appendix A: Dockerfile Used to Create Spark Container

```
# Dockerfile.spark.tf: Spark on CentOS container with TensorFlow and IoT Analytics Benchmark
# Based on Dockerfile.spark.centos

FROM centos

ARG spark_jars=jars
ARG img_path=kubernetes/dockerfiles

# Before building the docker image, first build and make a Spark distribution following
# the instructions in http://spark.apache.org/docs/latest/building-spark.html.
# Also need to copy a Java JDK to the spark directory (jdk-8u201-linux-x64.rpm used here)
# Modify /root/spark/kubernetes/dockerfiles/spark/entrypoint.sh to change last line to exec
"${CMD[@]}"
# If this docker file is being used in the context of building your images from a Spark
# distribution, the docker build command should be invoked from the top level directory
# of the Spark distribution. E.g.:
# docker build -t spark-tf:latest -f <path>/Dockerfile.spark.tf .

RUN set -ex
RUN mkdir -p /opt/spark/work-dir

COPY ${spark_jars} /opt/spark/jars
COPY bin /opt/spark/bin
COPY sbin /opt/spark/sbin
COPY ${img_path}/spark/entrypoint.sh /opt/
COPY jdk-8u201-linux-x64.rpm /opt

ENV SPARK_HOME /opt/spark

WORKDIR /
RUN mkdir ${SPARK_HOME}/python
RUN yum -y install iputils iproute which wget pdsh ntp nc unzip
RUN yum -y install https://centos7.iuscommunity.org/ius-release.rpm
RUN yum -y install python36u python36u-pip
RUN ln -s /usr/bin/python3.6 /usr/bin/python3
RUN ln -s /usr/bin/pip3.6 /usr/bin/pip3
RUN pip3 install --upgrade pip
RUN pip3 install numpy keras tensorflow
RUN rpm -iv /opt/jdk-8u201-linux-x64.rpm
ENV JAVA_HOME /usr/java/latest
RUN wget -nv https://github.com/vmware/iot-analytics-benchmark/archive/master.zip -O /root/iot-
analytics-benchmark-master.zip
RUN unzip /root/iot-analytics-benchmark-master.zip -d /root
RUN mkdir /root/BigDL
RUN wget -nv https://repo1.maven.org/maven2/com/intel/analytics/bigdl/dist-spark-2.4.0-scala-
2.11.8-all/0.8.0/dist-spark-2.4.0-scala-2.11.8-all-0.8.0-dist.zip -O /root/BigDL/dist-spark-
2.4.0-scala-2.11.8-all-0.8.0-dist.zip
RUN unzip /root/BigDL/dist-spark-2.4.0-scala-2.11.8-all-0.8.0-dist.zip -d /root/BigDL
RUN pip3 install BigDL==0.8.0
RUN mkdir -p /root/.keras/datasets
RUN wget -nv https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz -O
/root/.keras/datasets/cifar-10-batches-py.tar.gz
RUN cd /root/.keras/datasets; tar xzvf cifar-10-batches-py.tar.gz
RUN rm -r /root/.cache

COPY python/lib ${SPARK_HOME}/python/lib
ENV PYTHONPATH ${SPARK_HOME}/python/lib/pyspark.zip:${SPARK_HOME}/python/lib/py4j-*.zip

WORKDIR /opt/spark/work-dir

ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

References

[1] The Kubernetes Authors. (2019) Kubernetes (K8s) Production-Grade Container Orchestration. https://kubernetes.io

[2] VMware. (2019) Project Pacific. https://www.vmware.com/products/vsphere/projectpacific.html

[3] The Apache Software Foundation. (2018) Apache Spark - a unified analytics engine for large-scale data processing. https://spark.apache.org

[4] Dave Jaffe. (2017) Fast Virtualized Hadoop and Spark on All-Flash Disks: Best Practices for Optimizing Virtualized Big Data Applications on VMware vSphere 6.5. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/performance/bigdata-vsphere65-perf.pdf

[5] VMware. (2019) VMware Enterprise PKS: Deploy, run and manage Kubernetes for production. https://cloud.vmware.com/vmware-enterprise-pks

[6] VMware. (2019) VMware Cloud PKS. https://cloud.vmware.com/vmware-cloud-pks

[7] VMware. (2019) VMware Essential PKS. https://cloud.vmware.com/vmware-essential-pks

[8] Intel Analytics. (2019) GitHub: Intel Analytics BigDL. https://github.com/intel-analytics/BigDL

[9] Stanford Vision Lab, Stanford University, Princeton University. (2016) ImageNet. http://image-net.org/

[10] Dave Jaffe. (2019, August) GitHub: VMware IoT Analytics Benchmark. https://github.com/vmware/iot-analytics-benchmark

[11] MLPerf. (2019) MLPerf - Fair and useful benchmarks for measuring training and inference performance of ML hardware, software, and services. https://mlperf.org/

About the Author

Dave Jaffe is an engineer on the VMware Performance Engineering team, focusing on Big Data and Machine Learning.

Acknowledgements

Dave would like to thank Ralph Bankston of the VMware Essential PKS team for installation guidance, Louie Tsai of the Intel BigDL team for support, and Justin Murray of VMware Technical Marketing for continuing discussions of all things Big Data, Machine Learning, and Kubernetes.