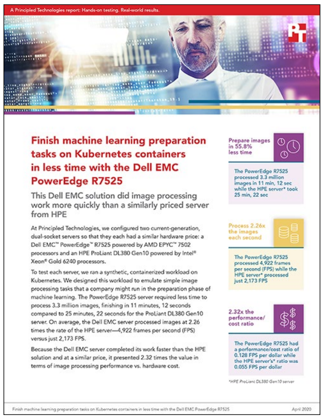
A Principled Technologies report: Hands-on testing. Real-world results.

Finish machine learning preparation tasks on Kubernetes containers in less time with the Dell EMC PowerEdge R7525

This Dell EMC solution did image processing work more quickly than a similarly priced server from HPE

At Principled Technologies, we configured two current-generation, dual-socket servers so that they each had a similar hardware price: a Dell EMC™ PowerEdge™ R7525 powered by AMD EPYC™ 7502 processors and an HPE ProLiant DL380 Gen10 powered by Intel® Xeon® Gold 6240 processors.

To test each server, we ran a synthetic, containerized workload on Kubernetes. We designed this workload to emulate simple image processing tasks that a company might run in the preparation phase of machine learning. The PowerEdge R7525 server required less time to process 3.3 million images, finishing in 11 minutes, 12 seconds compared to 25 minutes, 22 seconds for the ProLiant DL380 Gen10 server. On average, the Dell EMC server processed images at 2.26 times the rate of the HPE server—4,922 frames per second (FPS) versus just 2,173 FPS.

Because the Dell EMC server completed its work faster than the HPE solution and at a similar price, it presented 2.32 times the value in terms of image processing performance vs. hardware cost.

Prepare images in 55.8% less time

The PowerEdge R7525 processed 3.3 million images in 11 min, 12 sec while the HPE server* took 25 min, 22 sec

Process 2.26x the images each second

The PowerEdge R7525 processed 4,922 frames per second (FPS) while the HPE server* processed just 2,173 FPS

2.32x the performance/cost ratio

The PowerEdge R7525 had a performance/cost ratio of 0.128 FPS per dollar while the HPE server's* ratio was 0.055 FPS per dollar

*HPE ProLiant DL380 Gen10 server

Finish machine learning preparation tasks on Kubernetes containers in less time with the Dell EMC PowerEdge R7525 April 2020

The science behind the report:

Finish machine learning preparation tasks on Kubernetes containers in less time with the Dell EMC PowerEdge R7525

This document describes what we tested, how we tested, and what we found. To learn how these facts translate into real-world benefits, read the report Finish machine learning preparation tasks on Kubernetes containers in less time with the Dell EMC PowerEdge R7525.

We concluded our hands-on testing on March 12, 2020. During testing, we determined the appropriate hardware and software configurations and applied updates as they became available. The results in this report reflect configurations that we finalized on March 11, 2020 or earlier. Unavoidably, these configurations may not represent the latest versions available when this report appears.

Our results

Table 1: Time to prepare a set of 3.3 million images from our benchmark

	Dell EMC™ PowerEdge™ R7525	HPE ProLiant DL380 Gen10
Run 1	11 minutes, 10 seconds	25 minutes, 28 seconds
Run 2	11 minutes, 12 seconds	25 minutes, 21 seconds
Run 3	11 minutes, 15 seconds	25 minutes, 22 seconds
Median	**11 minutes, 12 seconds**	**25 minutes, 22 seconds**

Table 2: Data processing rate, hardware cost, and value

	Dell EMC PowerEdge R7525	HPE ProLiant DL380 Gen10
Processing rate in frames per second (FPS)	4,922	2,173
Hardware cost (USD)	$38,482.50	$39,846.00
Value (FPS per dollar)	0.128	0.055

CPU utilization for each server under test

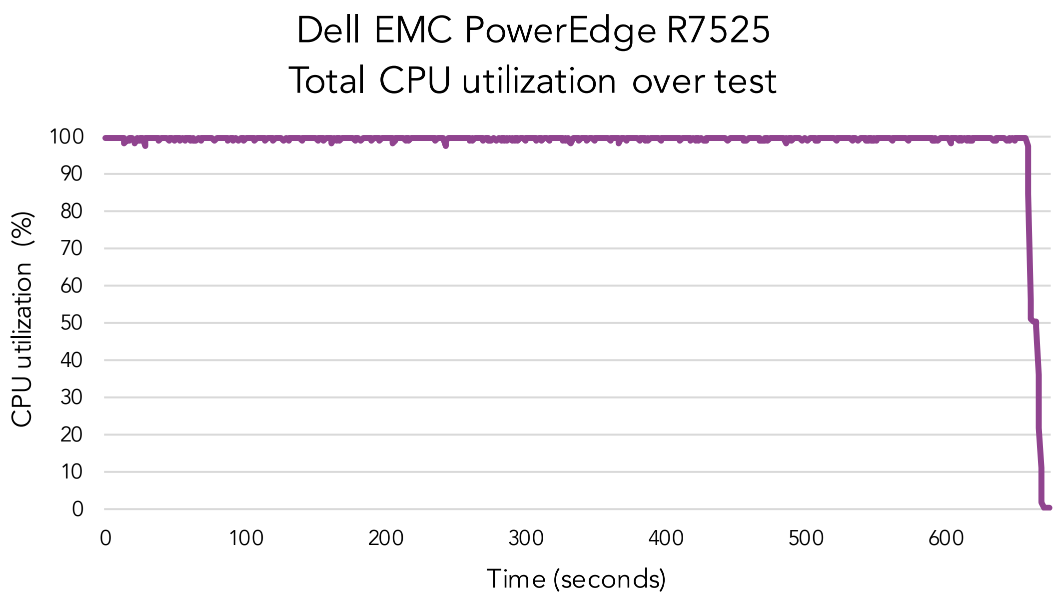

Figure 1: Graph of the Dell EMC PowerEdge R7525 server's CPU utilization for the duration of the machine learning preparation test. Source: Principled Technologies.

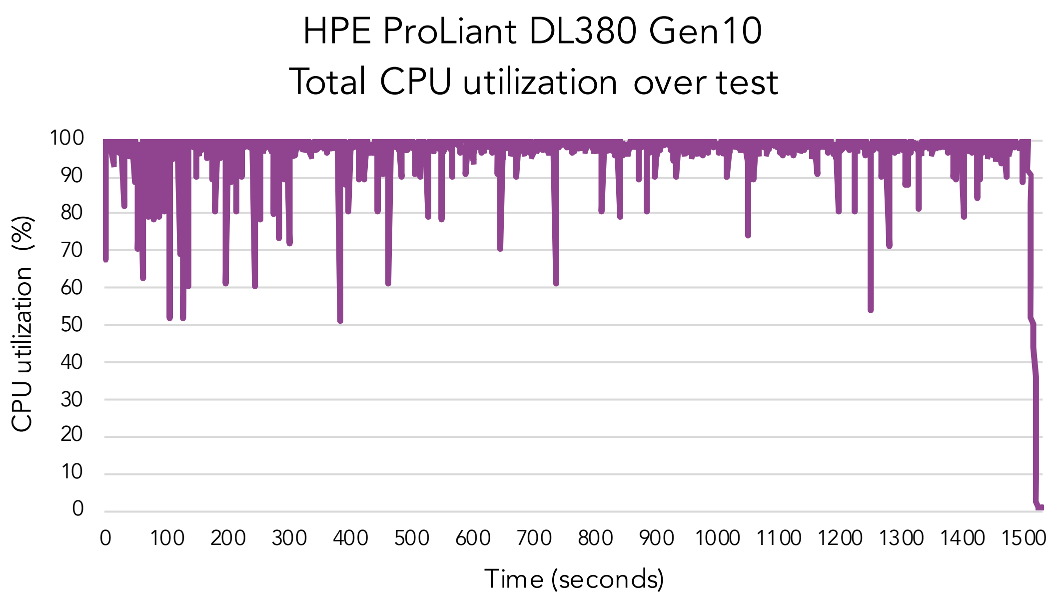

Figure 2: Graph of the HPE ProLiant DL380 Gen10 server's CPU utilization for the duration of the machine learning preparation test. Source: Principled Technologies.

System configuration information

Table 3: Detailed information on the systems we tested.

Server configuration information	Dell EMC PowerEdge R7525	HPE ProLiant DL380 Gen10
BIOS name and version	Dell 1.2.9	U30 v2.10
Non-default BIOS settings	System Profile Settings > System Profile: Performance	Workload Profile – High Performance Computer (HPC)
Operating system name and version/build number	Red Hat Enterprise Linux 8.1	Red Hat Enterprise Linux 8.1
Date of last OS updates/patches applied	02/07/2020	02/07/2020
Processor		
Number of processors	2	2
Vendor and model	AMD EPYC™ 7502	Intel® Xeon® Gold 6240
Core count (per processor)	32	18
Core frequency (GHz)	2.5	2.6
Stepping	U0	B1
Memory module		
Total memory in system (GB)	512	512
Number of memory modules	16	16
Vendor and model	SK Hynix HMA84GR7CJR4N-XN	SK Hynix HMA84GR7CJR4N-WM
Size (GB)	32	32
Type	PC4-3200	PC4-2933Y
Speed (MHz)	3,200	2,933
Speed running in the server (MHz)	3,200	2,933
Local storage		
Number of drives	5	5
Drive vendor and model	Samsung® PM1725b 1.6TB SFF	Samsung PM1725b 1.6TB SFF
Drive size (TB)	1.6	1.6
Drive information (speed, interface, type)	PCIe Gen 3 x4/dual x2	PCIe Gen 3 x4/dual x2
Software RAID usage	RAID10 using 4 disks	RAID10 using 4 disks
Network adapters		
Vendor and model	Broadcom Gigabit Ethernet BCM5720	HPE Ethernet 1Gb 4-port 331i Adapter
Number and type of ports	4 x 1GbE	4 x 1GbE
Cooling fans		
Vendor and model	DC Brushless PPPTM-X30	Nidec UltraFlo V60E12BS1M3
Number of cooling fans	6	6

Server configuration information	Dell EMC PowerEdge R7525	HPE ProLiant DL380 Gen10
Power supplies		
Vendor and model	Dell EMC D2400E-S1	HPE 865414-B21
Number of power supplies	2	2
Wattage of each (W)	2,400	800

How we tested

Overview

We installed Red Hat Enterprise Linux 8.1 on both servers and ran an image processing workload we created for preparation testing. We hosted the image dataset on a Linux software RAID10 we built from four NVMe drives.

Our environment

We prepared both servers identically except for the hardware differences we noted in our hardware disclosure (table 3, page 3). On each server, we installed Red Hat Enterprise Linux 8.1 using the minimal installation option with the development tools package to a single disk volume. During installation, we disabled kdump, enabled the Ethernet port, and changed the hostname to accommodate our environment. After installing Kubernetes, we deployed a pod of eight containers. Each container used a number of threads based on the number available for each processor. We then ran the preprocessing workload and measured the time required for the command to complete, as well as the frames per second each server processed.

We compared the servers using a purpose-built preprocessing workload we wrote in Python using publicly available open-source libraries. We define this workload in detail below. We can provide the code on request. Please contact info@principledtechnologies.com for more information.

Workload

Description

Our preparation workload emulates a simple image-processing workload by distributing dataset preparation tasks among M processes running on N nodes, the exact number of which is up to the user. Each process produces a single shard of the final data set by taking input images, performing simple conversions, encoding the resulting image, and appending it to the shard file.

This Python workload uses Pillow (https://python-pillow.org/) to perform image manipulations. The output format is a file with one Base64-encoded image per line.

We designed this application to operate in single-node mode or clustered mode, and added built-in logic for discovering cluster members when clustered. For this study, we used the application in single-node mode. We intended this preparation-stage application to be as computationally lightweight as possible. However, the pipeline is still compute-limited, even when using relatively slow storage.

This workload is particularly well suited to CPU comparisons with large thread/core-count disparity. Each thread operates independently, performs the same work, and is CPU-bound. Additional threads allow the server to complete more work per unit time, showing clear differentiation for higher core/thread count CPUs.

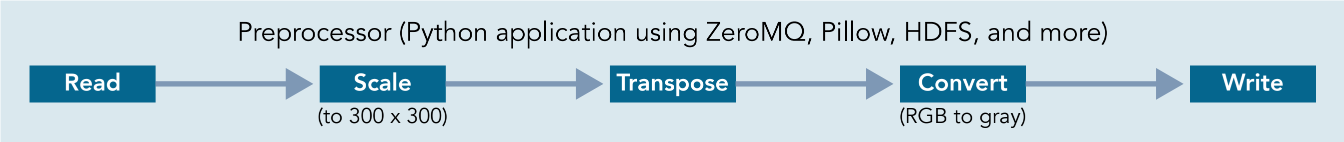

Figure 3: Pipeline data flow and operations. Source: Principled Technologies.

Figure 3 shows how the system reads images from storage, scales the images to a Machine-learning-friendly size of 300 x 300 pixels, transposes the images, and finally, converts the images to grayscale. The system then encodes the transformed image into JPG, and then converts the JPG-encoded to Base64 and writes it to the next line of the shard file. File systems for image-reading and shard-writing may be local storage or HDFS, though we used local storage for testing. In addition to the shard files, the workload also outputs frame count, byte count, frames per second, input and output bytes per second, and total runtime. In clustered mode, the workload also computes statistics across the cluster.

Application pseudocode

```
Let D = a directory containing JPG/PNG images
Let V = be the desired volume of data to process
Let N = number of nodes
Let M = number of processes per node

While not cluster achieved quorum: // (clustered mode only)
    Sleep // (clustered mode only)
```

```
V'=V/(M*N)

{launch M threads}:
Let selected = []
Let B_selected = 0 // number of bytes to write
For image in D:
    Selected.append(image)
    B_selected = B_selected + size(image)
    If B_selected > V'
            break
Let shard = open(output_file_name)
    For image in selected:
            Let image_resized = resize( image, [300,300] )
            Let image_transposed = transpose( image_resized )
            Let image_gray = rgb2gray( image_transposed )
            Let data = base64.encode( image_gray )
            shard.write(data+"\n")
    {compute thread statistics …}
{compute node statistics …}
Share statistics with cluster members…// (clustered mode only)
{compute cluster statistics …}// (clustered mode only)
```

Configuring our server for Kubernetes testing

1. After installing RHEL, use the subscription manager to register the operating system, update the software, and install mdadm and vim:

```
subscription-manager register --username * --password * --auto-attach
yum upgrade -y
yum install mdadm vim -y
```

2. Disable the firewall, and disable SELinux:

```
systemctl stop firewalld
systemctl disable firewalld
setenforce 0
#Edit the selinux config file
vi /etc/selinux/config
…
SELINUX = disabled
...
```

3. Prepare each of the four drives you need for the software RAID. We used lsblk to determine which drives to include. Perform the following commands on each individual disk:

```
parted
#Select the target disk
select /dev/nvme*n1
#Clear and create a new partition table.
mklabel gpt
#Create new primary partition
mkpart primary ext4 0 1.5T
```

4. Create the RAID10 using the following commands:

```
#Create a RAID10 from the 4 target NVME drive's partitions. List each of the target partitions for
each NVMe
mdadm --create /dev/md3 --level=10 --raid-devices=4 /dev/nvme*n1p1 /dev/nvme*n1p1 /dev/nvme*n1p1 /
dev/nvme*n1p1
#Define filesystem
mkfs.ext4 /dev/md3
#Mount the RAID
mkdir /stor
sudo mount /dev/md3 /stor
#add the disk to fstab so it mounts on reboot
vim /etc/fstab
/dev/md3 /stor ext4 defaults 0 2
```

5. Add the docker repository, and install docker-ce.

```
dnf config-manager --add-repo=https://download.docker.com/linux/centos/docker-ce.repo
dnf install docker-ce-17.12.1.ce-1.el7.centos
```

6. To install Kubernetes, first create the following repo file:

```
vim /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
               https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

7. To complete the Kubernetes installation, run the following commands:

```
yum update
yum install kubeadm
```

8. Restart and enable docker and kubelet services:

```
systemctl restart docker
systemctl enable docker
systemctl restart kubelet
systemctl enable kubelet
```

9. Initiate the Kubernetes cluster:

```
kubeadm init --apiserver-advertise-address=<Host IP address> --pod-network-cidr=192.168.0.0/16
mkdir -p $HOME/.kube
cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
chown $(id -u):$(id -g) $HOME/.kube/config
```

10. Taint the master node so you can deploy pods on it:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

11. Download the Kube-flannel yaml file:

```
wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
```

12. In the kube-flannel.yml file, change the net-conf.json network to 192.168.0.0/16
13. Create a file to build the pod. The following file creates a pod with eight containers. We divided the total CPU threads of each server by the number of containers, and set the number to be the CPU limit of each container. For the AMD EPYC 7502, the CPU limit is 16 and the CPU request is 8. For the Intel Xeon Gold 6242, the CPU limit is 9 and the CPU request is 5.

```
vi ai-pod-8containers.yml

apiVersion: v1
kind: Pod
metadata:
  name: ai-pod1
spec:
  containers:
  - name: ai1
    image: ptuser/preparation:latest
    ports:
      - containerPort: 80
    resources:
      requests:
        memory: "16384Mi"
        cpu: "8"
      limits:
        memory: "32768Mi"
        cpu: "16"
    volumeMounts:
      - mountPath: "/app"
        name: ai-app
```

```
      - mountPath: "/data"
        name: ai-data
      - mountPath: "/out"
        name: ai-out
  - name: ai2
    image: ptuser/preparation:latest
    ports:
      - containerPort: 80
    resources:
      requests:
        memory: "16384Mi"
        cpu: "8"
      limits:
        memory: "32768Mi"
        cpu: "16"
    volumeMounts:
      - mountPath: "/app"
        name: ai-app
      - mountPath: "/data"
        name: ai-data
      - mountPath: "/out"
        name: ai-out
  - name: ai3
    image: ptuser/preparation:latest
    ports:
      - containerPort: 80
    resources:
      requests:
        memory: "16384Mi"
        cpu: "8"
      limits:
        memory: "32768Mi"
        cpu: "16"
    volumeMounts:
      - mountPath: "/app"
        name: ai-app
      - mountPath: "/data"
        name: ai-data
      - mountPath: "/out"
        name: ai-out
  - name: ai4
    image: ptuser/preparation:latest
    ports:
      - containerPort: 80
    resources:
      requests:
        memory: "16384Mi"
        cpu: "8"
      limits:
        memory: "32768Mi"
        cpu: "16"
    volumeMounts:
      - mountPath: "/app"
        name: ai-app
      - mountPath: "/data"
        name: ai-data
      - mountPath: "/out"
        name: ai-out
  - name: ai5
    image: ptuser/preparation:latest
    ports:
      - containerPort: 80
    resources:
      requests:
        memory: "16384Mi"
```

```
          cpu: "8"
        limits:
          memory: "32768Mi"
          cpu: "16"
      volumeMounts:
        - mountPath: "/app"
          name: ai-app
        - mountPath: "/data"
          name: ai-data
        - mountPath: "/out"
          name: ai-out
    - name: ai6
      image: ptuser/preparation:latest
      ports:
        - containerPort: 80
      resources:
        requests:
          memory: "16384Mi"
          cpu: "8"
        limits:
          memory: "32768Mi"
          cpu: "16"
      volumeMounts:
        - mountPath: "/app"
          name: ai-app
        - mountPath: "/data"
          name: ai-data
        - mountPath: "/out"
          name: ai-out
    - name: ai7
      image: ptuser/preparation:latest
      ports:
        - containerPort: 80
      resources:
        requests:
          memory: "16384Mi"
          cpu: "8"
        limits:
          memory: "32768Mi"
          cpu: "16"
      volumeMounts:
        - mountPath: "/app"
          name: ai-app
        - mountPath: "/data"
          name: ai-data
        - mountPath: "/out"
          name: ai-out
    - name: ai8
      image: ptuser/preparation:latest
      ports:
        - containerPort: 80
      resources:
        requests:
          memory: "16384Mi"
          cpu: "8"
        limits:
          memory: "32768Mi"
          cpu: "16"
      volumeMounts:
        - mountPath: "/app"
          name: ai-app
        - mountPath: "/data"
          name: ai-data
        - mountPath: "/out"
          name: ai-out
```

```
volumes:
  - name: ai-app
    hostPath:
      path: /root/ai-pipeline-benchmarks
      type: Directory
  - name: ai-data
    hostPath:
      path: /stor/dataset
      type: Directory
  - name: ai-out
    emptyDir: {}
```

Deploying the Kubernetes Pods

Before each test run, we restarted our server under test. We waited 30 minutes, then kicked off the test by following the procedure below:

1. Turn off swap memory:

   ```
   swapoff -a
   ```

2. Build the pod network:

   ```
   kubectl apply -f kube-flannel.yml
   ```

3. Verify that the pods create successfully:

   ```
   kubectl get nodes
   kubectl config view
   kubectl get pods –all-namespaces
   ```

4. Create the pod and the containers:

   ```
   kubectl apply -f ai-pod-8containers.yml
   ```

5. To run the workload, navigate to the test code GIT repository, and find the testing/README.md. Follow the instructions, and use the settings described in the Workload section of this document. We ran the workload on each server three times. We reported the medium performance score in our final report.

Read the report at http://facts.pt/rfcwex2

This project was commissioned by Dell Technologies.

Principled Technologies®

Facts matter.®