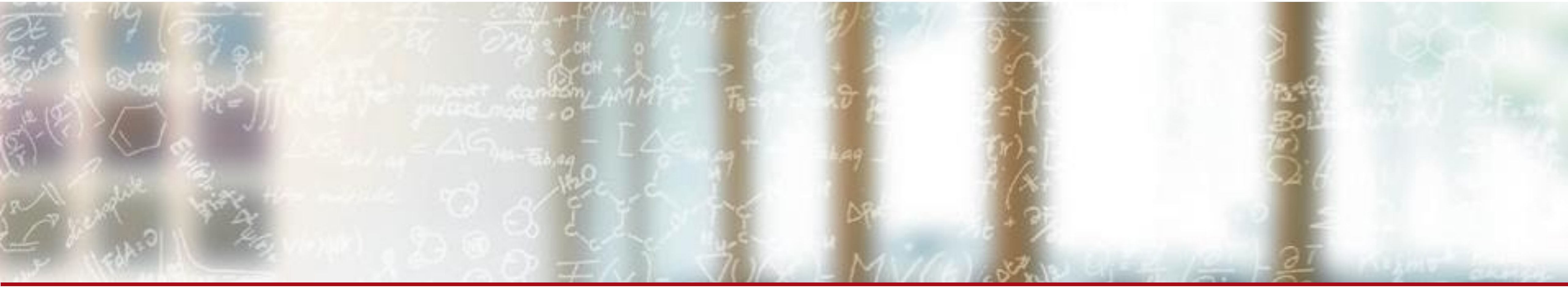
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



HPC Containers at CSCS – Building & Running

CSCS User Lab Day – Meet the Swiss National Supercomputing Center

Manitaras Theofilos-Ioannis, CSCS

September 2, 2022

Outline

- Introduction to Containers
- Building container images with Buildah & running with Sarus
- Building/Deploying/Running a GPU-enabled image
- Running MPI applications
- Conclusions

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction to Containers

Containers in a nutshell

What are Containers?

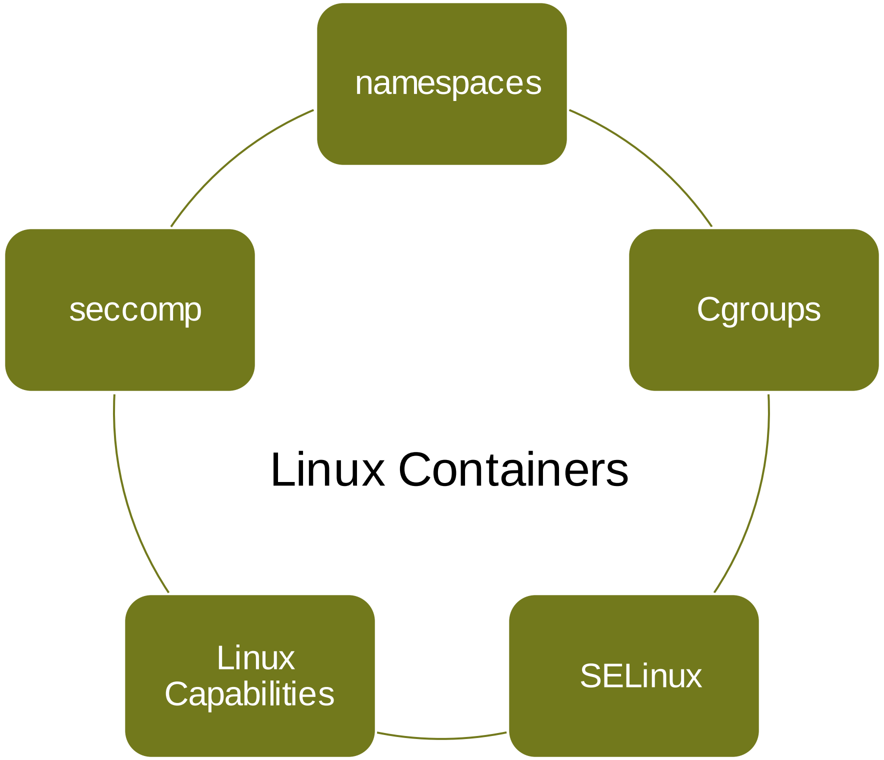
- Wikipedia (general container): *A container is any receptacle or enclosure for holding a product used in* ***storage****,* ***packaging****, and* ***shipping****. Things kept inside of a container are* ***protected*** *by being inside of its structure*
- Docker: *A container is a standard unit of software that* ***packages up code and all its dependencies,*** *so the application runs quickly and reliably from one computing environment to another*
- Google Cloud: *Containers offer a* ***logical packaging mechanism*** *in which applications can be abstracted from the environment in which they actually run*
- AWS: *Containers are a method of* ***operating system virtualization*** *that allow you to run an application and its dependencies in* ***resource-isolated processes***

Linux Containers under the hood

- Linux namespaces: Control what the process (container) can "see"
- cgroups: limit and monitor the resources that the process (container) can use

Security

- SELinux: Security-Enhanced Linux
- Linux capabilities: restrict allowed syscalls
- seccomp: Secure Computing



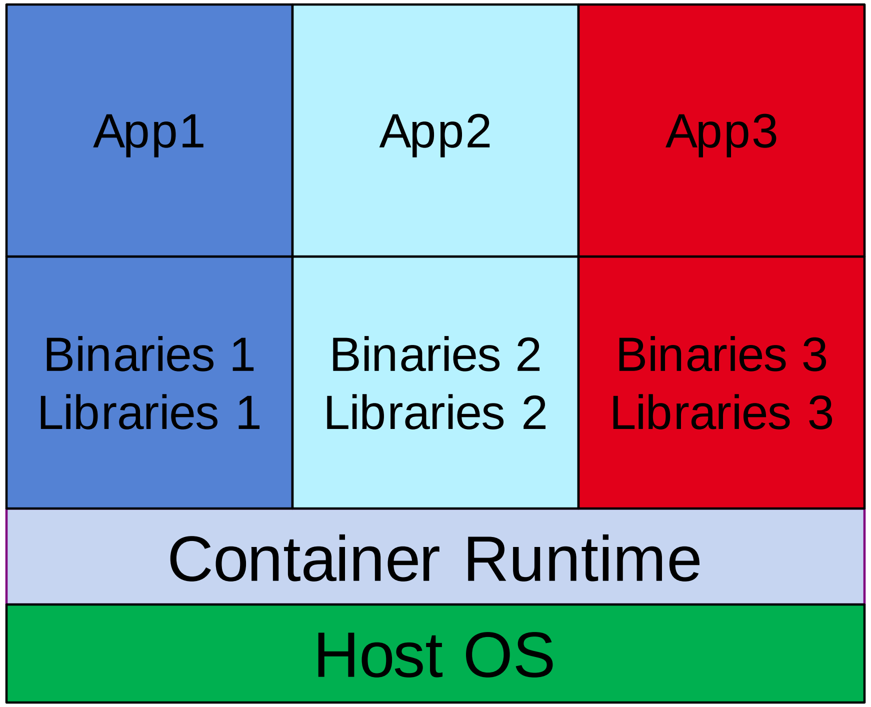
Linux Namespaces

The Linux namespaces used in containers are:

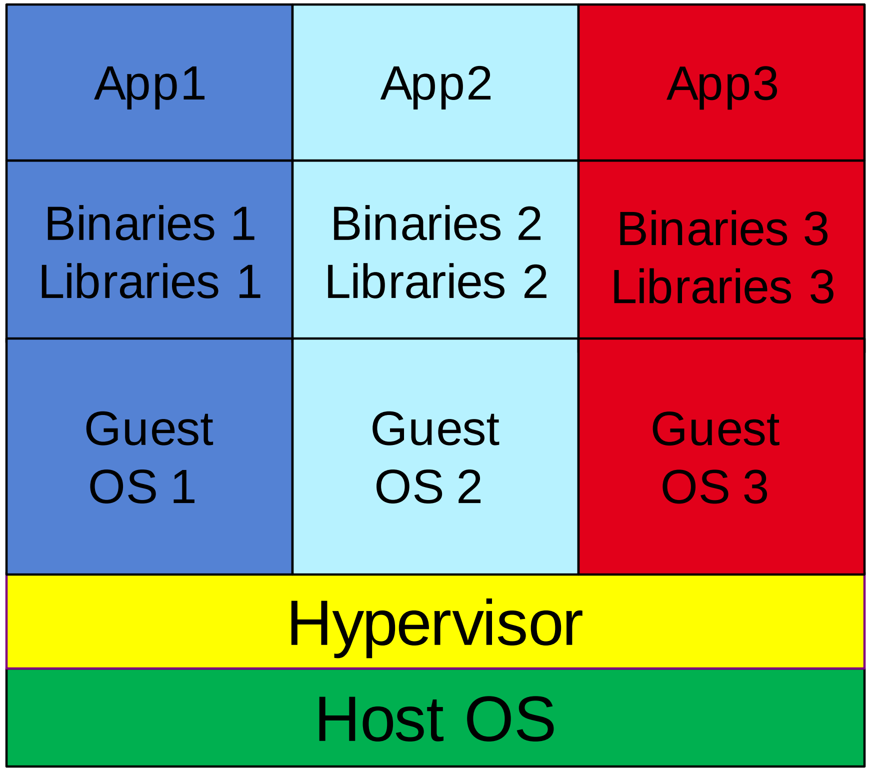
- Mount namespace
- UTS namespace
- IPC namespace
- Network namespace
- Pid namespace
- User namespace

- There are additional namespaces and new ones might be introduced

Containers vs Virtual Machines



Containers

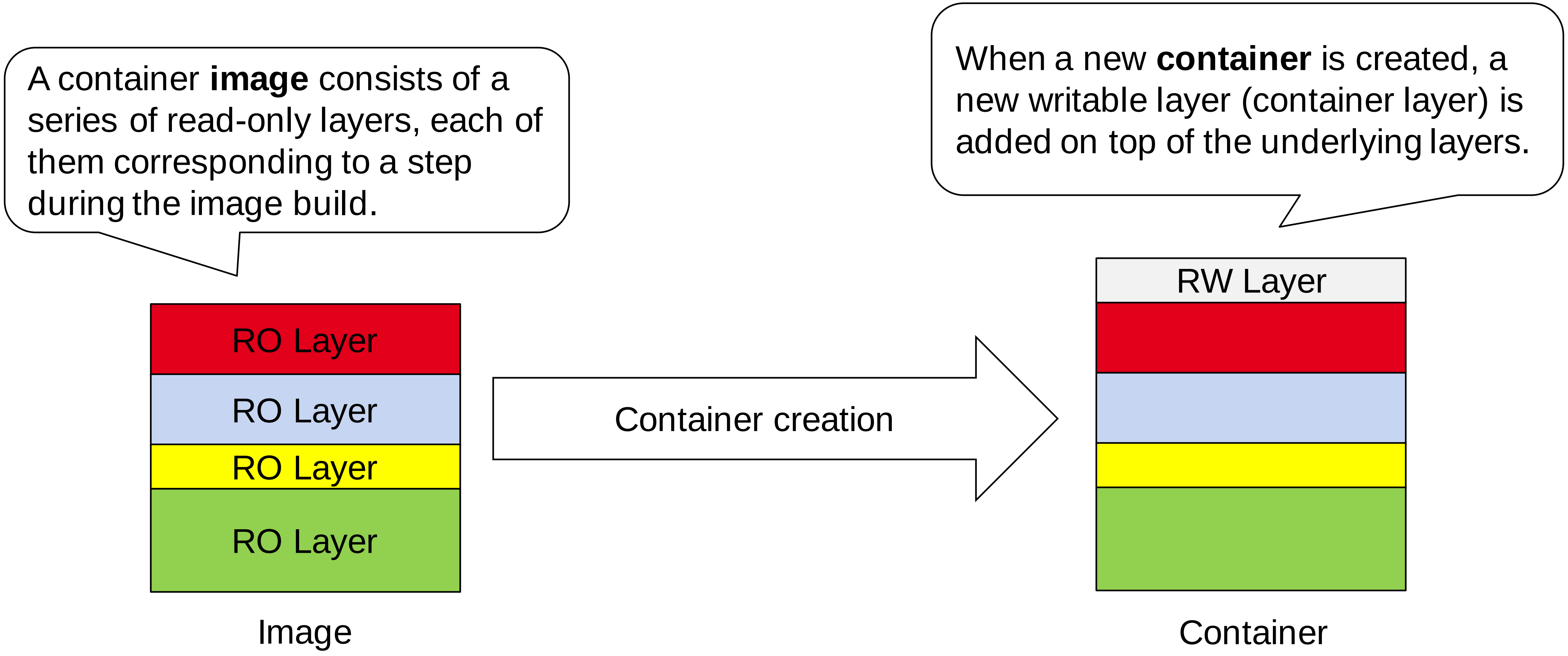


Virtual Machines

Container advantages

- Bundle software and dependencies in a single portable package
- Address the requirement of different versions of a library by individual packages
- Faster shipping of software updates
- Easier, predicable and automation friendly deployment
- Effortless upgrade of software components
- Reproducible behavior in different computing environments
- Container adoption promotes development of loosely coupled software components
- Fast start-up (milliseconds-few seconds vs seconds-minutes for vms)
- Isolation of software components and control over their communication

Container images vs running containers



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Building container images with Buildah & running with Sarus

Buildah in a Nutshell

- Buildah is a tool that facilitates building OCI container images
- Supports building images either by modifying existing ones, or via Containerfiles
- Can run rootless and is daemonless
- It's OCI-compliant, allows importing/exporting images in different archive types
- Works with both public & private container registries
- Allows spawning containers from images, executing commands interactively
- It's actively developed, with new features being added

Containerfiles and the build context

- **Containerfiles/Dockerfiles** are text files containing the necessary instructions needed to build a container image
- Containerfiles consist of a series of **steps** written in a syntax, with each step adding a new layer to the image
- The default name for a Containerfile expected by Buildah is **"Containerfile/Dockerfile"**
- To build a container image using a Containerfile a **build context** is also needed
- The **build context** is the set of files located at the specified PATH or URL and is used by Buildah during the build of the image filesystem

Containerfile steps (1/2)

The Containerfile consists of a series of steps the most important of which are:

1. **FROM <base_image>:** specifies the base image that is going to be used during the build. This step is mandatory and is the first step of each Containerfile.
2. **RUN <command> [params]:** runs the given shell commands. Multiple RUN commands are allowed inside a Containerfile
3. **ENV <key> <value>:** sets an environment variable to a given value. Multiple environment variables can be set and ENV can be used multiple times
4. **WORKDIR <workdir>:** specifies the working directory in which subsequent commands are going to be executed.
5. **USER <user>:** change the user from this step and beyond
6. **ARG <key>[=<default>]:** define a variable to be used during building

Dockerfile steps (2/2)

7. **CMD <cmd>:** the command the container is going to run if not instructed otherwise
8. **ENTRYPOINT <entrypoint> [params]:** used to configure a container that will run as an executable
9. **COPY <source> <destination>:** copies files from the build context to the image filesystem
10. **ADD <source> <destination>:** like COPY but additionally it handles archives and URLS and provides extra options
11. **VOLUME <path>:** specifies a volume to be used when running a container from this image
12. **EXPOSE <port>:** is used as a label to inform about the exposed ports of the container

Build container images using Buildah on Piz Daint (1/2)

User documentation is available here

1. Create a job allocation using the "contbuild" constraint:
   salloc -C'gpu&contbuild' -A <project> -N1 -n1 --time=480
2. SSH to the compute node allocated in step 1:
   ssh $SLURM_NODELIST
3. Load the required modules:
   module load daint-gpu Buildah
4. Set up a storage configuration file under **~/.config/containers/storage.conf**, containing the following:

```
[storage]
  driver = "vfs"
  runroot = "/scratch/local/<username>/runroot"
  graphroot = "/scratch/local/<username>/root"
```

Build container images using Buildah on Piz Daint (2/2)

5. Building container images using Dockerfiles is straightforward by using the command "buildah bud" (build using Dockerfile):
   buildah bud -f <Dockerfile_name> -t <image_tag> .
6. A container can be spawned from a given image using:
   buildah from --name=<container_name> <image_name>
7. Run interactively inside the container:
   buildah run -t <container_name> bash
8. Commit the changes made to the container as a new image:
   buildah commit <container_name> <new_image_tag>
9. Tag and push to an image registry or export as a tar archive:
   buildah tag <new_image_tag> <registry/organization/repository:tag>
   buildah push <registry/organization/repository:tag>
   buildah push <new_image_tag> docker-archive:<path_to_arhive>

Limitations of Buildah at CSCS

- User namespaces do not work well with nfs-type filesystems (see here). Therefore, an ext4 partition is needed to have sufficient space to build the images (mounted when using the 'contbuild' Slurm constraint)
- Before the latest Piz Daint os upgrade, only the vfs storage driver was supported since fuse-overlayfs needs a kernel version >= 4.18 (see here) *
- The ext4 partition is cleared and unmounted after the allocated job finishes thus, you have to make sure you either export your images to an archive or push them to a container registry

* The fuse-overlayfs support is being tested at the moment and will be available soon, offering better performance and requiring less storage space per image

Sarus in a Nutshell

SARUS

- Sarus is a software to run Linux containers on HPC environments
- Security oriented to HPC systems
- Extensible runtime by means of OCI hooks to allow current and future support of custom hardware while achieving native performance
- Creation of container filesystems tailored for diskless nodes and parallel filesystems
- Compatibility with the presence of a workload manager
- Compatibility with the Open Container Initiative (OCI) standards

Build-Tag-Login-Push-Pull-Run

1. Build a container image on Piz Daint using Buildah, based on a Containerfile:

   buildah bud --format=docker -t <image_name> .

2. Tag the image based on the registry, the repository and the image name / tag:

   buildah tag <image_name> <registry/repository/name:tag>

3. Login to the container registry using credentials:

   buildah login <registry>

4. Push the image to the registry:

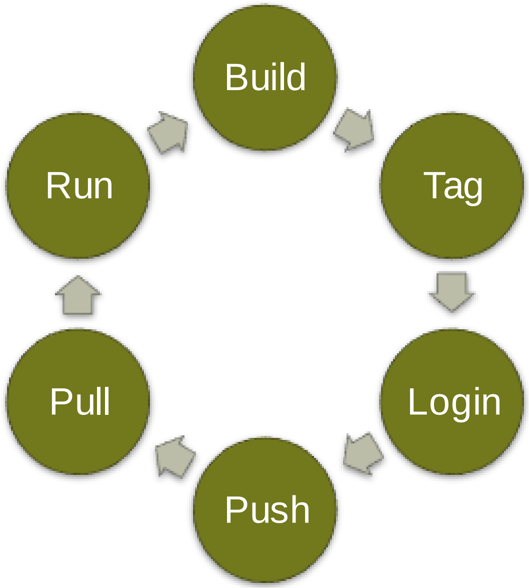
   buildah push <registry/repository/name:tag>

5. Pull the image using Sarus:

   sarus pull --login <registry/repository/name:tag>

6. Run a container based on the image with Sarus:

   sarus run <registry/repository/name:tag>



Building/Deploying/Running a GPU-enabled image

Containerfile for Cuda Device Query

```
FROM nvidia/cuda:11.2.1-devel-ubuntu20.04

ARG SMS=60

RUN apt-get update && \
    apt-get install -y git && \
    git clone https://github.com/NVIDIA/cuda-samples.git /usr/local/cudasamples && \
    cd /usr/local/cudasamples && \
    git fetch origin --tags && \
    git checkout v11.0 && \
    make -C Samples/deviceQuery -j$(nproc)

FROM nvidia/cuda:11.2.1-runtime-ubuntu20.04

COPY --from=0 /usr/local/cudasamples/bin/x86_64/linux/release/deviceQuery /cuda_samples/deviceQuery

CMD /cuda_samples/deviceQuery
```

Build the Cuda Device Query image with Buildah

1. Create a job allocation, land on the compute node and load the corresponding modules (as documented earlier)
2. Change to the directory with the Container file and build the image:
   buildah bud –t cuda-device-query:11.2.1 -f Containerfile .
3. Tag the corresponding image according to the registry of choice:
   buildah tag cuda-device-query:11.2.1 <registry url>/cuda-device-query:11.2.1
4. Login to the registry of choice:
   buildah login <registry url>
5. Push the image to the registry:
   buildah tag cuda-device-query:11.2.1 <registry url>/cuda-device-query:11.2.1
6. Alternatively, the image can be saved as a tar archive on scratch:
   buildah push cuda-device-query:11.2.1 docker-archive:$SCRATCH/cuda-device-query.tar

Load and run with Sarus

1. Pull the image from the registry using Sarus:
   srun -C gpu -A <project> --pty sarus pull –login <registry_url>/cuda-device-query:11.2.1
2. Alternatively, the image can be loaded from the tar archive:
   srun -C gpu -A <project> sarus load $SCRATCH/cuda-device-query.tar <target_name>:<target_tag>
3. Run using Sarus:
   srun -C gpu -A <project> sarus run <registry_url>/cuda-device-query:11.2.1

- Container images for GPU-accelerated applications must feature an installation of the CUDA Toolkit
- No direct user interaction is required to make GPUs available inside the container
- Sarus handles the GPU interaction via NVIDIA Container Runtime hook

Running the container with Sarus

```
 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "Tesla P100-PCIE-16GB"
  CUDA Driver Version / Runtime Version          11.4 / 11.2
  CUDA Capability Major/Minor version number:    6.0
  Total amount of global memory:                 16281 MBytes (17071734784 bytes)
  (56) Multiprocessors, ( 64) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1329 MHz (1.33 GHz)
  Memory Clock rate:                             715 Mhz
  Memory Bus Width:                              4096-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 2 / 0
  Compute Mode:
     < Exclusive Process (many threads in one process is able to use ::cudaSetDevice() with this device) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime Version = 11.2, NumDevs = 1
Result = PASS
```

Running MPI applications

Containerfile for OSU-benchmark

```
FROM debian:jessie

RUN apt-get update && \
    apt-get install -y ca-certificates file g++ gcc gfortran make gdb strace realpath wget --no-install-recommends

RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz && \
    tar xf mpich-3.1.4.tar.gz && \
    cd mpich-3.1.4 && ./configure --disable-fortran --enable-fast=all,O3 --prefix=/usr && \
    make -j$(nproc) && make install && ldconfig

RUN wget -q http://mvapich.cse.ohio-state.edu/download/mvapich/osu-micro-benchmarks-5.3.2.tar.gz && \
    tar xf osu-micro-benchmarks-5.3.2.tar.gz && \
    cd osu-micro-benchmarks-5.3.2 && \
    ./configure --prefix=/usr/local CC=$(which mpicc) CFLAGS=-O3 && \
    make && make install && cd .. && \
    rm -rf osu-micro-benchmarks-5.3.2 && \
    rm osu-micro-benchmarks-5.3.2.tar.gz

CMD /usr/local/libexec/osu-micro-benchmarks/mpi/pt2pt/osu_bw
```

Containerfile for OSU-benchmark using Spack (1/2)

```
FROM ubuntu:20.04 as builder

RUN apt-get update && apt-get install -y \
      build-essential curl gcc git gfortran python3 libpmi2-0-dev && \
      rm -rf /var/lib/apt/lists/*

RUN git clone -b v0.18.1 https://github.com/spack/spack.git

RUN mkdir /opt/spack-environment \
&& (echo "spack:" \
&&  echo "  specs:" \
&&  echo "  - osu-micro-benchmarks target=haswell ^mpich@3.4.3 pmi=pmi2" \
&&  echo "  concretizer:" \
&&  echo "    unify: true" \
&&  echo "  config:" \
&&  echo "    install_tree:"\
&&  echo "      root: /opt/software" \
&&  echo "  view: /opt/view") > /opt/spack-environment/spack.yaml

RUN . /spack/share/spack/setup-env.sh && \
      spack compiler find && \
      cd /opt/spack-environment && \
      spack env activate . && \
      spack install -v --fail-fast && \
      spack gc -y
```

Containerfile for OSU-benchmark using Spack (2/2)

```
RUN . /spack/share/spack/setup-env.sh && \
      cd /opt/spack-environment && \
      spack env activate --sh -d . >> /etc/profile.d/z10_spack_environment.sh  && \
      spack env activate . && \
      spack gc -y && \
      # Add the mpich lib path to ld.so.conf.d directory
      spack find --format='{prefix.lib}' mpich > /etc/ld.so.conf.d/mpich.conf

FROM ubuntu:20.04

RUN apt-get update && apt-get install -y \
   libpmi2-0-dev libatomic1 ibgfortran5 && \
   rm -rf /var/lib/apt/lists/*

COPY --from=builder /opt/spack-environment /opt/spack-environment
COPY --from=builder /opt/software /opt/software
COPY --from=builder /opt/view /opt/view
COPY --from=builder /etc/profile.d/z10_spack_environment.sh /etc/profile.d/z10_spack_environment.sh
COPY --from=builder /etc/ld.so.conf.d/mpich.conf /etc/ld.so.conf.d/mpich.conf

RUN ldconfig

ENTRYPOINT ["/bin/bash", "--rcfile", "/etc/profile", "-l", "-c"]
```

Load and run with Sarus

1. Pull the image from the registry using Sarus:
   srun -C gpu -A <project> --pty sarus pull --login <registry_url>/osu_mpich:gcc
2. Run using the container MPI:
   srun -N2 -C gpu -A <project> --mpi=pmi2 sarus run <registry_url>/osu_mpich:gcc
3. Run using the Sarus MPI hook:
   srun **-N2 -C gpu -A <project> sarus run --mpi <registry_url>/osu_mpich:gcc**

- To run containers using the MPI implementation embedded in the image, the host system process manager is still used (hybrid approach)
- In order to access the high-speed Cray Aries interconnect, the container application must be dynamically linked to an MPI implementation that is ABI-compatible with the compute node's MPI =
- This is where the MPI hook of Sarus becomes crucial to get the best performance

Running with/without the MPI hook

Run with Sarus MPI hook

```
# OSU MPI Bandwidth Test v5.9
# Size      Bandwidth (MB/s)
1                       1.55
2                       3.14
4                       6.33
8                      12.74
16                     25.44
32                     51.39
64                    102.63
128                   203.22
256                   402.39
512                   785.23
1024                 1157.23
2048                 1831.73
4096                 2481.02
8192                 6335.60
16384                8534.92
32768                9242.51
65536                9537.90
131072               9723.74
262144               9820.70
524288               9867.66
1048576              9881.81
2097152              9891.75
4194304              9866.98
```

Run with container MPI

```
# OSU MPI Bandwidth Test v5.9
# Size      Bandwidth (MB/s)
1                       0.18
2                       0.36
4                       0.72
8                       1.44
16                      2.88
32                      5.72
64                     11.31
128                    22.27
256                    28.81
512                    57.25
1024                   73.85
2048                  159.00
4096                  309.92
8192                  619.77
16384                1170.05
32768                2280.48
65536                3784.08
131072               5256.85
262144               6349.60
524288               7278.74
1048576              7523.48
2097152              7575.73
4194304              7596.65
```

Conclusions

Conclusions

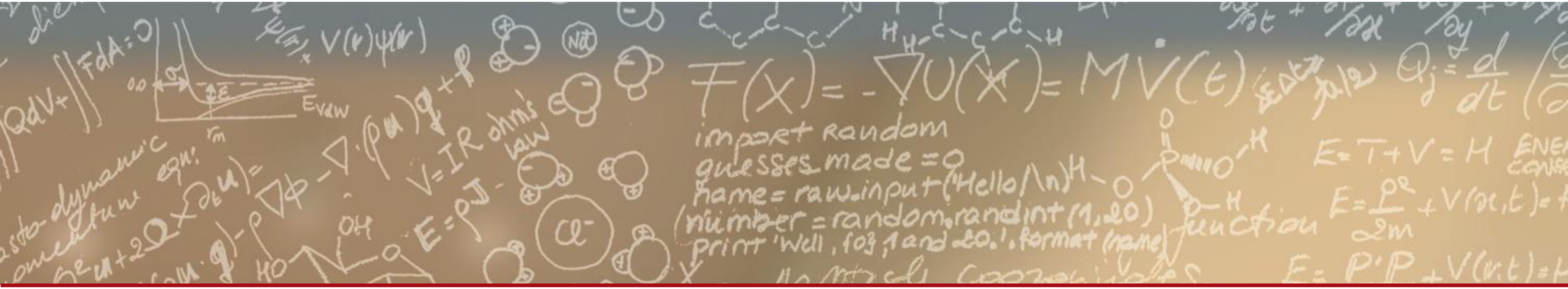
- Containers allow bundling software + dependencies in a single package
- Containers offer portability of software between multiple systems
- Fast start-up (milliseconds-few seconds vs seconds-minutes for VMs)
- Buildah can be effectively used on CSCS systems to build container images
- Produced images can be pulled/loaded with Sarus and run at scale
- Running an gpu-enabled image is straightforward with Sarus
- MPI-ABI compatibility has to be ensured to use the Sarus MPI hook and get the best performance
- Using the above applications, you can containerize your software and run at scale on the CSCS systems

Additional Resources

- Buildah web page
- Buildah GitHub repository
- Buildah documentation at CSCS
- Sarus GitHub repository
- Sarus documentation at CSCS
- Sarus Cookbook

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Thank you for your attention.