GPU Computing
Beyond video games
## Infrastructure at NVIDIA

**DGX SuperPOD GPU cluster (Top500 #20)**

### SYSTEM SPECIFICATIONS

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPUs</td>
<td>16X NVIDIA® Tesla V100</td>
</tr>
<tr>
<td>GPU Memory</td>
<td>512GB total</td>
</tr>
<tr>
<td>Performance</td>
<td>2.1 petaFLOPS</td>
</tr>
<tr>
<td>NVIDIA CUDA® Cores</td>
<td>81920</td>
</tr>
<tr>
<td>NVIDIA Tensor Cores</td>
<td>10240</td>
</tr>
<tr>
<td>NVSwitches</td>
<td>12</td>
</tr>
<tr>
<td>Maximum Power Usage</td>
<td>12kW</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel® Xeon® Platinum 8174</td>
</tr>
<tr>
<td></td>
<td>CPU @3.10GHz, 24 cores per CPU</td>
</tr>
<tr>
<td>System Memory</td>
<td>1.5TB</td>
</tr>
<tr>
<td>Network</td>
<td>8X 100Gb/sec</td>
</tr>
<tr>
<td></td>
<td>Infiniband/100GigE</td>
</tr>
<tr>
<td></td>
<td>Dual 10/25/40/50/100GbE</td>
</tr>
<tr>
<td>Storage</td>
<td>OS: 2X 960GB NVME SSDs</td>
</tr>
<tr>
<td></td>
<td>Internal Storage: 30TB</td>
</tr>
<tr>
<td></td>
<td>(8X 3.84TB) NVME SSDs</td>
</tr>
<tr>
<td>Software</td>
<td>Ubuntu Linux OS</td>
</tr>
</tbody>
</table>

The TOP500 project ranks and details the 500 most powerful non-distributed computer systems in the world. The project was started in 1993 and publishes an updated list of the supercomputers twice a year. [Wikipedia](https://en.wikipedia.org/wiki/TOP500)

**Start date:** 1993
NVIDIA Containers
Supports all major container runtimes

We built *libnvidia-container* to make it easy to run CUDA applications inside containers.

We release optimized container images for each of the major Deep Learning frameworks every month.

We use containers for everything on our HPC clusters - R&D, official benchmarks, etc.

Containers give us portable software stacks without sacrificing performance.
Typical cloud deployment

e.g. Kubernetes

Hundreds/thousands of small nodes
All applications are containerized, for security reasons
Many small applications running per node (e.g. microservices)
Traffic to/from the outside world
Not used for interactive applications or development
Advanced features: rolling updates with rollback, load balancing, service discovery
GPU Computing at NVIDIA

HPC-like

10-100 very large nodes

“Trusted” users

Not all applications are containerized

Few applications per node (often just a single one)

Large multi-node jobs with checkpointing (e.g. Deep Learning training)

Little traffic to the outside world, or air-gapped

Internal traffic is mostly RDMA
Slurm Workload Manager

https://slurm.schedmd.com/slurm.html

Advanced scheduling algorithms (fair-share, backfill, preemption, hierarchical quotas)
Gang scheduling: scheduling and starting all processes of a multi-node job simultaneously
Low runtime overhead
Topology-aware (NUMA/PCIe) job scheduling for better performance
Simple CLI with jobs as bash scripts
GPUs are a first-class resource
Supports interactive jobs

Slurm does not support containers out of the box... but is extensible through plugins
Containers for HPC

What do we need?

- High performance
- Support for Docker images
- Soft cluster multi-tenancy
- Exposing NVIDIA GPUs and Mellanox InfiniBand cards inside containers
- Resources (CPU/Mem/GPUs/HCAs) isolation through cgroups
- Launching multi-node jobs
- Development workflow: interactive jobs, installing packages, debugging

No existing container runtime fulfilled all our requirements, so we built our own
Unprivileged runtime
aka “rootless”

Writing a secure privileged container runtime is very hard (see the latest runc CVEs)
Watch "Securing Container Runtimes -- How Hard Can It Be?" - Aleksa Sarai (LCA 2020)

Even when trusting users to not actively exploit the runtime, we don’t want real root:
- users could break the system, or corrupt shared filesystems
- users won’t be able to delete files created from the container
- users won’t be able to gdb/strace applications running inside the container
ENROOT
Overview

Fully unprivileged “chroot”
Standalone
Little isolation, no overhead
Docker image support
Simple image format
Composable and extensible
Simple and easy to use
Advanced features
User namespaces
root outside container != root inside container

We use “user namespaces” with optional remapping to UID 0

```bash
root@superpod-01:/root# cat /proc/self/uid_map
  0   1000  1

felix@superpod-01:/home/felix$ cat /proc/self/uid_map
1000  1000  1
```

Some applications refuse to run as UID 0
Convenient to have the same username and $HOME inside and outside the container
runc-based container runtimes always remap you to UID 0 inside the container
Subordinate UIDs/GIDs
/etc/subuid and /etc/subgid

We run application containers, we don’t need UID separation.

Installing packages requires to be UID 0, plus additional UIDs.

Difficult to maintain subordinate UIDs across multiple nodes.

Permissions issues for files you created while assuming a subordinate UID.

We use a seccomp filter to trap all setuid-related syscalls, to make them succeed.
Standalone runtime
Low overhead and ephemeral

No persistent spawning daemon

Inherits cgroups from the job as opposed to Docker

The runtime prepares the container and then executes the application

runc and docker (containerd-shim) have “supervisor” processes
Minimal isolation
Containers for packaging applications, not sandboxing

We don’t need an IP for each container, nor need to bind “privileged” ports

A PID namespace requires careful handling for the PID 1 process and tend to confuse programs

We want only 2 namespaces: mount and user

Resource isolation (cgroups) is handled by the scheduler (e.g. Slurm)

Having minimal isolation simplifies the runtime and improves performance
Impact on performance
Container isolation is bad for performance

Using a network namespace adds overhead (bridge, NAT, overlay...)

Seccomp and LSMs (AppArmor, SELinux) have an overhead

We need a shared IPC namespace (and /dev/shm) for fast intra-node communications

Rlimits might not be adapted to certain workloads (e.g. Docker memlock)

Seccomp triggers Spectre mitigation on most distributions:

```
$ docker run ubuntu grep 'Speculation_Store_Bypass' /proc/self/status
Speculation_Store_Bypass: thread force mitigated
```
Message Passing Interface
The HPC industry standard

We use MPI for intra/inter nodes communications of distributed jobs

PID/IPC namespaces confuses MPI for intra-node communications

CMA (process_vm_writev) requires ptrace access between processes

We use PMI/PMIx for coordination and need pass file descriptors from Slurm to the application
Importing Docker images

Speeding up the pull

The hardest part of the container runtime: authentication, OCI manifests, AUFS whiteouts

Rely on overlayfs rather than sequential extraction (e.g. docker, umoci)

Pipelines like “parallel curl|pigz|tar” tend to be faster than Golang alternatives

The “vfs” format has a huge storage cost (each layer copies the full rootfs)

Layers are usually uncompressed and not shared across users

Enroot shares layers across users, and they are compressed with zstd

We have helper binaries with capabilities to convert AUFS to overlayfs and to create a squashfs image of all the layers
Image format
KISS and Unixy

Standard squashfs file and configuration files:
ENTRYPOINT=/etc/rc     ENV=/etc/environment     VOLUME=/etc/fstab

Editing configuration from within the container is straightforward

Squashfs images, can be stored on parallel filesystems as a single file

Avoids thundering herd problems on multi-node jobs

Useful for air gapped systems, admins can control the applications you can run

Can be mounted as a block device and lazily fetched (e.g. over NFS)
Simple and Extensible
Accommodates heterogeneous clusters

The runtime is a simple shell script consisting of ~500 LoC

Uses a set of basic Linux utilities for unprivileged users

System wide and user-specific configurations to control the container environment, mounts and (prestart) hooks

Admins and users can customize the runtime, including tweaking builtins features (e.g. cgroups, shadow DB, GPU/HCA support)
ENROOT

Basic usage

# Convert a Docker image to squashfs file
$ enroot import docker://nvcr.io#nvidia/tensorflow:19.08-py3
$ ls nvidia+tensorflow+19.08-py3.sqsh

# Extract a squashfs to a rootfs
$ enroot create --name tensorflow nvidia+tensorflow+19.08-py3.sqsh
$ ls -d ${XDG_DATA_PATH}/enroot/tensorflow

# Start the container with optional root remapping and read/write rootfs
$ enroot start tensorflow nvidia-smi -L
$ enroot start --root --rw tensorflow apt update && apt install …
# Run an in-memory container from a squashfs image through fuse
$ enroot start ubuntu.sqsh

# Build a self-extracting TensorFlow bundle (image + runtime) and run it like you
# would run any executable
$ enroot bundle --output tensorflow.run nvidia+tensorflow+19.05-py3.sqsh
$ ./tensorflow.run python -c 'import tensorflow as tf; print(tf.__version__)'
ENROOT
Advanced Linux utilities

enroot-unshare : similar to unshare(1) and nsenter(1)
enroot-mount : similar to mount(8)
enroot-switchroot : similar to pivot_root(8) and login(1)
enroot-aufs2ovlfs : converts AUFS whiteouts to OverlayFS
enroot-mksquashovlfs : mksquashfs(1) on top of OverlayFS
ENROOT
“Container” from scratch

$ curl https://cdimage.ubuntu.com/ [...] /ubuntu-base-16.04-core-amd64.tar.gz | tar -C ubuntu -xz

$ enroot-nsenter --user --mount bash

$ cat << EOF | enroot-mount --root ubuntu -
  ubuntu  / none bind,rprivate
  /proc   /proc none rbind
  /dev    /dev none rbind
  /sys    /sys none rbind
EOF

$ exec enroot-switchroot ubuntu bash
Slurm plugin

100% YAML-free

Our plugin adds new arguments to the Slurm CLI:

# Bare-metal application
$ srun python train.py

# Containerized application
$ srun --container-image=tensorflow/tensorflow python train.py

The container image is imported and the container is started in the background before Slurm execs python train.py, the plugin joins the running container:

- setns(container_userns_fd, CLONE_NEWUSER)
- setns(container_mntns_fd, CLONE_NEWNS)
- import /proc/container_pid/environ
- chdir(container_workdir)
Slurm example 1
Interactive single-node job

# You can start a container by importing an image from Docker Hub
$ srun --container-name=pytorch --container-image=pytorch/pytorch apt install -y vmtouch

# We can reuse the existing container and execute our new binary on the dataset
$ srun --container-name=pytorch --container-mounts=/mnt/datasets:/datasets vmtouch /datasets

# Now we can start an interactive session inside the container, for development/debugging:
$ srun --container-name=pytorch --container-mounts=/mnt/datasets:/datasets --pty bash
root@superpod-01:/workspace#
$ cat job.sh
#!/bin/bash

# You can start a container on each node from a shared squashfs file
srun --nodes=64 --ntasks-per-node=16 --mpi=pmix \
    --container-image=/mnt/apps/tensorflow.sqsh --container-mounts=/mnt/datasets:/datasets \
    python train.py /datasets/cats

# The container rootfs get automatically cleaned up at the end of the script
echo "Training complete!"

# Submitting the job is done through the command-line too
$ sbatch --nodes=64 --output=tensorflow_training.log job.sh
We built a new container runtime for our use case

a. Unprivileged
b. Lightweight, without excessive isolation
c. Flexible plugins, including support for NVIDIA and Mellanox devices

http://github.com/nvidia/enroot
http://github.com/nvidia/pyxis (Slurm plugin)

Thanks to our colleagues: Julie Bernauer, Louis Capps, Michael Knox, Luke Yeager