NGC CONTAINER

DU-08812-001 _v03 | October 2019

User Guide

TABLE OF CONTENTS

Chapter 1. DOCKER CONTAINERS

Over the last few years there has been a dramatic rise in the use of software containers for simplifying deployment of data center applications at scale. Containers encapsulate an application along with its libraries and other dependencies to provide reproducible and reliable execution of applications and services without the overhead of a full virtual machine.

GPU support within Docker containers enables GPU-based applications that are portable across multiple machines in a similar way to how Docker® enables CPU-based applications to be deployed across multiple machines.

Docker container

A Docker container is an instance of a Docker image. A Docker container deploys a single application or service per container.

Docker image

A Docker image is simply the software (including the filesystem and parameters) that you run within a nvidia-docker container.

1.1. What Is A Docker Container?

A Docker container is a mechanism for bundling a Linux application with all of its libraries, data files, and environment variables so that the execution environment is always the same, on whatever Linux system it runs and between instances on the same host.

Unlike a VM which has its own isolated kernel, containers use the host system kernel. Therefore, all kernel calls from the container are handled by the host system kernel. DGX™ systems uses Docker containers as the mechanism for deploying deep learning frameworks.

A Docker container is the running instance of a Docker image.

1.2. Why Use A Container?

One of the many benefits to using containers is that you can install your application, dependencies and environment variables one time into the container image; rather than on each system you run on. In addition, the key benefits to using containers also include:

- Install your application, dependencies and environment variables one time into the container image; rather than on each system you run on.
- There is no risk of conflict with libraries that are installed by others.
- Containers allow use of multiple different deep learning frameworks, which may have conflicting software dependencies, on the same server.
- After you build your application into a container, you can run it on lots of other places, especially servers, without having to install any software.
- Legacy accelerated compute applications can be containerized and deployed on newer systems, on premise, or in the cloud.
- Specific GPU resources can be allocated to a container for isolation and better performance.
- You can easily share, collaborate, and test applications across different environments.
- Multiple instances of a given deep learning framework can be run concurrently with each having one or more specific GPUs assigned.
- Containers can be used to resolve network-port conflicts between applications by mapping container-ports to specific externally-visible ports when launching the container.

Chapter 2. NGC IMAGES

NGC containers are hosted in a repository called **`nvcr.io`**. As you read in the previous section, these containers can be “pulled” from the repository and used for GPU accelerated applications such as scientific workloads, visualization, and deep learning.

A Docker image is simply a file-system that a developer builds. A Docker image serves as the template for the container, and is a software stack that consists of several layers. Each layer depends on the layer below it in the stack.

From the Docker image, a container is formed. When creating a container, you add a writable layer on top of the stack. A Docker image with a writable container layer added to it is a container. A container is simply a running instance of that image. All changes and modifications made to the container are made to the writable layer. You can delete the container; however, the Docker image remains untouched.

The NVIDIA Container Toolkit allows users to build and run GPU accelerated Docker containers. The toolkit includes a container runtime library and utilities to automatically configure containers to leverage NVIDIA GPUs.

Figure 1 depicts the DGX software stack. Notice that the NVIDIA Container Toolkit sits above the host OS and the NVIDIA Drivers. The tools are used to create and use NVIDIA containers - these are the layers above the cinainerization tool layer. These containers have applications, deep learning SDKs, and the CUDA® Toolkit™ . The NVIDIA containerization tools take care of mounting the appropriate NVIDIA Drivers.

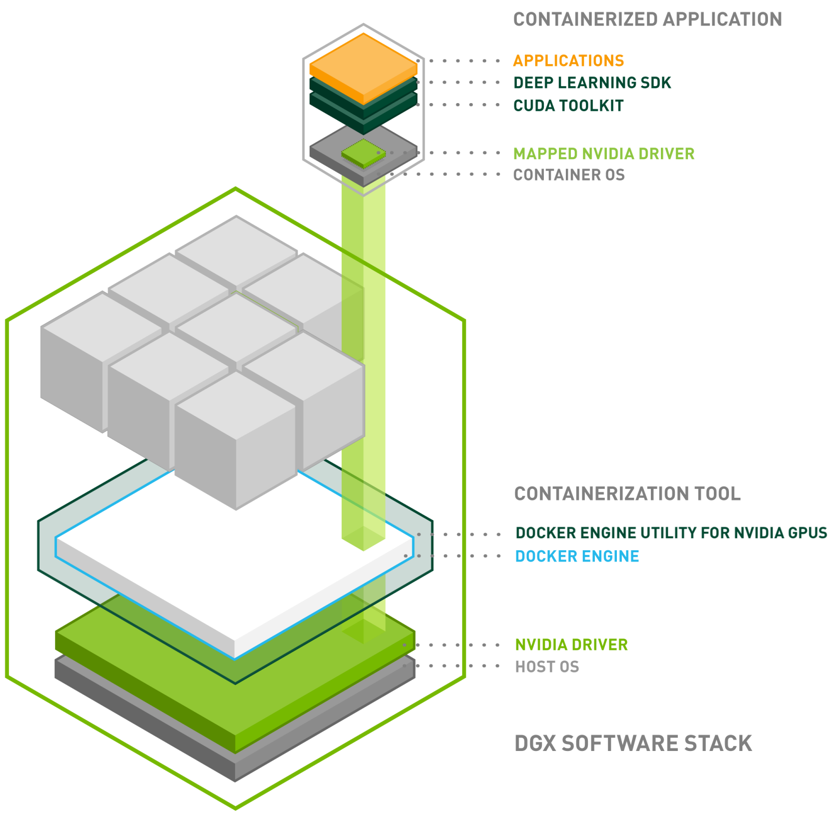
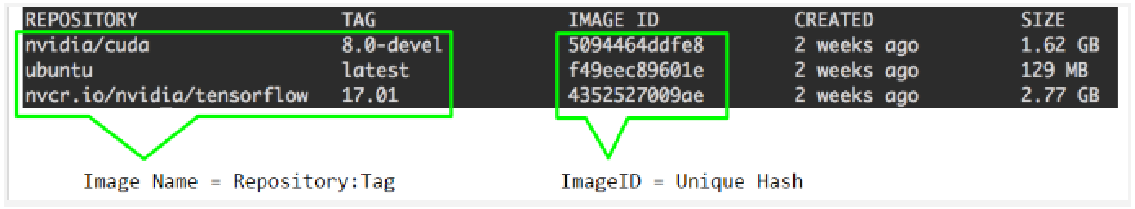


Figure 1 nvidia-docker Utilities within the NVIDIA Contrial Tool mount the user mode components of the NVIDIA driver and the GPUs into the Docker container at launch.

2.1. NGC Image Versions

Each release of an NGC image is identified by a version “tag”. For simpler images this version tag usually contains the version of the major software package in the image. More complex images which contain multiple software packages or versions may use a separate version solely representing the containerized software configuration. One common scheme is versioning by the year and month of the image release. For example, the 17.01 release of an image was released in January, 2017.

An image name consists of two parts separated by a colon. The first part is the name of the container in the repository and the second part is the “tag” associated with the container. These two pieces of information are shown in Figure 2, which is the output from issuing the **`docker images`** command.



```
REPOSITORY                TAG          IMAGE ID       CREATED        SIZE
nvidia/cuda               8.0-devel    5094464ddfe8   2 weeks ago    1.62 GB
ubuntu                    latest       f49eec89601e   2 weeks ago    129 MB
nvcr.io/nvidia/tensorflow 17.01        4352527009ae   2 weeks ago    2.77 GB
```

Figure 2 Output from **`docker images`** command

Figure 2 shows simple examples of image names, such as:

- **`nvidia-cuda:8.0-devel`**
- **`ubuntu:latest`**
- **`nvcr.io/nvidia/tensorflow:17.01`**

If you choose not to add a tag to an image, by default the word "latest " is added as the tag, however all NGC containers have an explicit version tag.

In the next sections, you will use these image names for running containers. Later in the document, there is also a section on creating your own containers or customizing and extending existing containers.

2.2. CUDA Toolkit Container

The CUDA Toolkit provides a development environment for creating high performance GPU-accelerated applications. The toolkit includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler and a runtime library to deploy your application.

All NGC Container images are based on the CUDA platform layer (**`nvcr.io/nvidia/cuda`**). This image provides a containerized version of the software development stack underpinning all other NGC containers, and is available for users who need more flexibility to build containers with custom CUDA applications.

2.2.1. OS Layer

Within the software stack, the lowest layer (or base layer) is the user space of the OS. The software in this layer includes all of the security patches that are available within the month of the release.

2.2.2. CUDA Layer

Compute Unified Device Architecture® (CUDA) is a parallel computing platform and programming model created by NVIDIA to give application developers access to the massive parallel processing capability of GPUs. CUDA is the foundation for GPU acceleration of deep learning as well as a wide range of other computation and memory-intensive applications ranging from astronomy, to molecular dynamics simulation, to computational finance.

2.2.2.1. CUDA Runtime

The CUDA runtime layer provides the components needed to execute CUDA applications in the deployment environment. The CUDA runtime is packaged with the CUDA Toolkit and includes all of the shared libraries, but none of the CUDA compiler components.

2.2.2.2. CUDA Toolkit

The CUDA Toolkit provides a development environment for developing optimized GPU-accelerated applications. It includes GPU-accelerated CUDA libraries which enable drop-in acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics. For developing custom algorithms, you can use available integrations with commonly used languages and numerical packages as well as well-published development APIs.

Chapter 3. NGC CONTAINER REGISTRY SPACES

The NGC container registry uses spaces to group NGC image repositories for related applications. These spaces appear in the image URL as a **`nvcr.io/<space>/image-name:tag`**, when used in pulling, running, or layering additional software on top of NGC container images.

`nvcr.io/nvidia`

This space contains a catalog of fully integrated and optimized deep learning framework containers that take full advantage of NVIDIA GPUs in both single GPU and multi-GPU configurations. They include CUDA Toolkit, DIGITS workflow, and the following deep learning frameworks: NVCaffe, Caffe2, Microsoft Cognitive Toolkit (CNTK), MXNet, PyTorch, TensorFlow, Theano, and Torch. These framework containers are delivered ready-to-run, including all necessary dependencies such as CUDA runtime, NVIDIA libraries, and an operating system.

Each framework container image also includes the framework source code to enable custom modifications and enhancements, along with the complete software development stack.

NVIDIA updates these deep learning containers monthly to ensure they continue to provide peak performance.

`nvcr.io/nvidia-hpcvis`

This space contains a catalog of HPC visualization containers, currently available in beta, featuring the industry's leading visualization tools, including ParaView with NVIDIA IndeX volume renderer, NVIDIA Optix ray-tracing library and NVIDIA Holodeck for interactive real-time visualization and high-quality visuals.

`nvcr.io/hpc`

This space contains a catalog of popular third-party GPU ready HPC application container provided by partners, including GAMESS, GROMACS, LAMMPS, NAMD and RELION. All third-party containers conform to NGC container standards and best practices, making it easy to get the latest GPU optimized HPC software up and get running quickly.

Chapter 4. PREREQUISITES

To enable portability in Docker images that leverage GPUs, three methods of providing GPU support for Docker containers have been developed.

- Native GPU support (preferred method if running Docker-ce 19.03 or later)
- **`nvidia-docker2`**
- **`nvidia-docker`**

Each of these methods mount the user mode components of the NVIDIA driver and the GPUs into the Docker container at launch. They allow NGC containers to take full advantage of NVIDIA GPUs.

- DGX™ users should follow the instructions in the Preparing to use NVIDIA Containers Getting Started Guide.
- NGC Cloud Images available from NGC Cloud Service Providers are set up with all the software needed to run NGC containers.
- User running NGC containers on TITAN PCs, Quadro PCs, or vGPUs should follow the corresponding instructions.
- Other users should follow the **`nvidia-docker`** installation documentation at **`nvidia-docker`** installation and install the latest NVIDIA Display driver for your GPU product type and series for your operating system. If NVIDIA drivers are not already configured on your system, then install them from here: Download Drivers.
- Ensure you have an NVIDIA GPU supporting Compute Unified Device Architecture® (CUDA) version with compute capability 6.0.0 or higher. For example, Pascal GPU architecture generation or later.
- Log into the NVIDIA® GPU Cloud (NGC) Container Registry located at **`nvcr.io`** using your NGC API key. For step-by-step instructions on how to gain access and get your API key, see NGC Getting Started Guide.

HPC Visualization Containers have different prerequisites. For more information, see #unique_14.

Chapter 5. PULLING A CONTAINER

Before you using a container from the NGC container registry, review the prerequisites described in the previous chapter.

You must also have access and logged into the NGC container registry as explained in the NGC Getting Started Guide.

5.1. Key Concepts

In order to issue the pull and run commands, ensure that you are familiar with the following concepts.

A pull command looks similar to:

```
docker pull nvcr.io/nvidia/caffe2:17.10
```

A run command looks similar to:

```
docker run --gpus all -it --rm –v local_dir:container_dir nvcr.io/nvidia/
caffe2:<xx.xx>
```

The base command **`docker run --gpu all`** assumes that your system has Docker 19.03-CE installed. See the section Enabling GPU Support for NGC Containers for the command to use for earlier versions of Docker.

The following concepts describe the separate attributes that make up the both commands.

`nvcr.io`

The name of the container registry, which for the NGC container registry and the NVIDIA DGX container registry is **`nvcr.io`**.

`nvidia`

The name of the space within the registry that contains the container. For containers provided by NVIDIA, the registry space is **`nvidia`**. For more information, see NGC Container Registry Spaces.

`-it`
You want to run the container in interactive mode.

`--rm`
You want to delete the container when finished.

`-v`
You want to mount the directory.

`local_dir`
The directory or file from your host system (absolute path) that you want to access from inside your container. For example, the **`local_dir`** in the following path is **`/home/jsmith/data/mnist`**.

```
-v /home/jsmith/data/mnist:/data/mnist
```

If you are inside the container, for example, using the command **`ls /data/mnist`**, you will see the same files as if you issued the ls **`/home/jsmith/data/mnist`** command from outside the container.

`container_dir`
The target directory when you are inside your container. For example, **`/data/mnist`** is the target directory in the example:

```
 -v /home/jsmith/data/mnist:/data/mnist
```

`<xx.xx>`
The tag. For example, 17.10.

5.2. Accessing And Pulling From The NGC container registry

You can access the NGC container registry by running a Docker commands from any Linux computer with Internet access on which Docker is installed.You can access the NGC container registry at **`nvcr.io`** through the Docker CLI.

Before accessing the NGC container registry, see NGC Getting Started Guide for instructions on how to access the website and, if you intend to access locked NGC content, know how to sign up for an NGC account and obtain an API key.

5.2.1. Logging in to the NGC container registry

Before accessing locked NGC content, you must sign up for an NGC account and obtain an API key as explained in the NGC Getting Started Guide. Then log in to the NGC registry from the command line as follows.

1. Log in to the NGC container registry.

```
$ docker login nvcr.io
```

2. When prompted for your user name, enter the following text:

```
$oauthtoken
```

3. The **$oauthtoken** username is a special user name that indicates that you will authenticate with an API key and not a username and password.
4. When prompted for your password, enter your NGC API key as shown in the following example.

```
Username: $oauthtoken
Password: k7cqFTUvKKdiwGsPnWnyQFYGnlAlsCIRmlP67Qxa
```

Tip When you get your API key, copy it to the clipboard so that you can paste the API key into the command shell when you are prompted for your password.

5.2.2. Pulling A Container From The NGC container registry Using The Docker CLI

You can browse the available containers in the NGC container registry by opening the NGC website using a web browser

1. Pull the container that you want from the registry. For example, to pull the NAMD container:

```
$ docker pull nvcr.io/hpc/namd:2.13
```

   You can copy the correct pull command for a specific NGC container by browsing the NGC website, selecting an NGC container, then clicking the download icon. Paste the command into the command line. See the NGC Getting Started Guide for details on using the NGC website.
2. List the Docker images on your system to confirm that the container was pulled.

```
$ docker images
```

   For more information pertaining to your specific container, refer to the **/workspace/README.md** file inside the container.

After pulling a container, you can run jobs in the container to run scientific workloads, train neural networks, deploy deep learning models, or perform AI analytics.

Chapter 6. RUNNING A CONTAINER

Before you can run an NGC deep learning framework container, your Docker environment must support NVIDIA GPUs. To run a container, issue the appropriate command as explained in this chapter, specifying the registry, repository, and tags.

6.1. Enabling GPU Support for NGC Containers

To obtain the best performance when running NGC containers, three methods of providing GPU support for Docker containers have been developed:

- Native GPU support (included with Docker-ce 19.03 or later)
- NVIDIA Container Runtime for Docker (**`nvidia-docker2`** package)
- Docker Engine Utility for NVIDIA GPUs (**`nvidia-docker`** package)

The method implemented in your system depends on the DGX OS version installed (for DGX systems), the specific NGC Cloud Image provided by a Cloud Service Provider, or the software that you have installed in preparation for running NGC containers on TITAN PCs, Quadro PCs, or vGPUs.

Refer to the following table to assist in determining which method is implemented in your system.

GPU Support Method	When Used	How to Determine
Native GPU Support	Included with Docker-ce 19.03 or later	Run **`docker version`** to determine the installed Docker version.
NVIDIA Container Runtime for Docker	If the **`nvidia-docker2`** package is installed	Run **`nvidia-docker version`** and check for NVIDIA Docker version 2.0 or later
Docker Engine Utility for NVIDIA GPUs	If the **`nvidia-docker`** package is installed	Run **`nvidia-docker version`** and check for NVIDIA Docker version 1.x

Each method is invoked by using specific Docker commands, described as follows.

Using Native GPU support

If Docker is updated to 19.03 on a system which already has nvidia-docker or nvidia-docker2 installed, then the corresponding methods can still be used.

- To use the native support on a new installation of Docker, first enable the new GPU support in Docker.

  ```
  $ sudo apt-get install -y docker nvidia-container-toolkit
  ```

  This step is not needed if you have updated Docker to 19.03 on a system with **`nvidia-docker2`** installed. The native support will be enabled automatically.
- Use **`docker run --gpus`** to run GPU-enabled containers.
  - Example using all GPUs

    ```
    $ docker run --gpus all ...
    ```
  - Example using two GPUs

    ```
    $ docker run --gpus 2 ...
    ```
  - Examples using specific GPUs

    ```
    $ docker run --gpus "device=1,2" ...
    $ docker run --gpus "device=UUID-ABCDEF,1" ...
    ```

Using the NVIDIA Container Runtime for Docker

With the NVIDIA Container Runtime for Docker installed (**`nvidia-docker2`**), you can run GPU-accelerated containers in one of the following ways.

- Use **`docker run`** and specify **`runtime=nvidia`**.

  ```
  $ docker run --runtime=nvidia ...
  ```
- Use **`nvidia-docker run`**.

  ```
  $ nvidia-docker run ...
  ```

  The new package provides backward compatibility, so you can still run GPU-accelerated containers by using this command, and the new runtime will be used.
- Use **`docker run`** with **`nvidia`** as the default runtime.

  You can set **`nvidia`** as the default runtime, for example, by adding the following line to the **`/etc/docker/daemon.json`** configuration file as the first entry.

  ```
  "default-runtime": "nvidia",
  ```

  The following is an example of how the added line appears in the JSON file. Do not remove any pre-existing content when making this change.

  ```
  {
   "default-runtime": "nvidia",
    "runtimes": {
       "nvidia": {
           "path": "/usr/bin/nvidia-container-runtime",
           "runtimeArgs": []
       }
   },
  ```

```
}
```

You can then use **docker run** to run GPU-accelerated containers.

```
$ docker run ...
```

Caution If you build Docker images while **nvidia** is set as the default runtime, make sure the build scripts executed by the Dockerfile specify the GPU architectures that the container will need. Failure to do so may result in the container being optimized only for the GPU architecture on which it was built. Instructions for specifying the GPU architecture depend on the application and are beyond the scope of this document. Consult the specific application build process for guidance.

Using the Docker Engine Utility for NVIDIA GPUs

With the Docker Engine Utility for NVIDIA GPUs installed (**nvidia-docker**), run GPU-enabled containers as follows.

```
$ nvidia-docker run ...
```

6.2. Running NGC Containers

On a system with GPU support for NGC containers, the following occurs when running a container.

- The Docker Engine loads the image into a container which runs the software.
- You define the runtime resources of the container by including additional flags and settings that are used with the command. These flags and settings are described in the following sections.
- The GPUs are explicitly defined for the Docker container (defaults to all GPUs, can be specified using *NV_GPU* environment variable).

The base command **docker run --gpu all** assumes that your system has Docker 19.03-CE installed. See the section Enabling GPU Support for NGC Containers for the command to use for earlier versions of Docker.

1. As a user, run the container interactively.

```
$ docker run --gpus all -it --rm –v local_dir:container_dir
         nvcr.io/nvidia/<repository>:<xx.xx>
```

The following example runs the December 2016 release (16.12) of the NVCaffe container in interactive mode. The container is automatically removed when the user exits the container.

```
$ docker run --gpus all --rm -ti nvcr.io/nvidia/caffe:16.12

===========
== Caffe ==
===========
NVIDIA Release 16.12 (build 6217)

Container image Copyright (c) 2016, NVIDIA CORPORATION.  All rights
 reserved.
Copyright (c) 2014, 2015, The Regents of the University of California
 (Regents)
All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION.  All rights
 reserved.
NVIDIA modifications are covered by the license terms that apply to the
 underlying project or file.
root@df57eb8e0100:/workspace#
```

2. From within the container, start the job that you want to run.

   The precise command to run depends on the deep learning framework in the container that you are running and the job that you want to run. For details see the **/workspace/README.md** file for the container.

   The following example runs the **caffe time** command on one GPU to measure the execution time of the **deploy.prototxt** model.

   ```
   # caffe time -model models/bvlc_alexnet/ -solver deploy.prototxt -gpu=0
   ```

3. Optional: Run the December 2016 release (16.12) of the same NVCaffe container but in non-interactive mode.

   ```
   % docker run --gpus all --rm nvcr.io/nvidia/caffe:16.12 caffe time -model
         /workspace/models/bvlc_alexnet -solver /workspace/deploy.prototxt -
   gpu=0
   ```

6.3. Specifying A User

Unless otherwise specified, the user inside the container is the root user.

When running within the container, files created on the host operating system or network volumes can be accessed by the root user. This is unacceptable for some users and they will want to set the ID of the user in the container. For example, to set the user in the container to be the currently running user, issue the following:

```
% docker run --gpus all -ti --rm -u $(id -u):$(id -g) nvcr.io/nvidia/
<repository>:<tag>
```

Typically, this results in warnings due to the fact that the specified user and group do not exist in the container. You might see a message similar to the following:

```
groups: cannot find name for group ID 1000I have no name! @c177b61e5a93:/
workspace$
```

The warning can usually be ignored.

6.4. Setting The Remove Flag

By default, Docker containers remain on the system after being run. Repeated pull or run operations use up more and more space on the local disk, even after exiting the container. Therefore, it is important to clean up the nvidia-docker containers after exiting.

Do not use the **--rm** flag if you have made changes to the container that you want to save, or if you want to access job logs after the run finishes.

To automatically remove a container when exiting, add the **--rm** flag to the run command.

```
% docker run --gpus all --rm nvcr.io/nvidia/<repository>:<tag>
```

6.5. Setting The Interactive Flag

By default, containers run in batch mode; that is, the container is run once and then exited without any user interaction. Containers can also be run in interactive mode as a service.

To run in interactive mode, add the **-ti** flag to the run command.

```
% docker run --gpus all -ti --rm nvcr.io/nvidia/<repository>:<tag>
```

6.6. Setting The Volumes Flag

There are no data sets included with the containers, therefore, if you want to use data sets, you need to mount volumes into the container from the host operating system. For more information, see Manage data in containers.

Typically, you would use either Docker volumes or host data volumes. The primary difference between host data volumes and Docker volumes is that Docker volumes are private to Docker and can only be shared amongst Docker containers. Docker volumes are not visible from the host operating system, and Docker manages the data storage. Host data volumes are any directory that is available from the host operating system. This can be your local disk or network volumes.

Example 1

Mount a directory **/raid/imagedata** on the host operating system as **/images** in the container.

```
% docker run --gpus all -ti --rm -v /raid/imagedata:/images
        nvcr.io/nvidia/<repository>:<tag>
```

Example 2

Mount a local docker volume named **data** (must be created if not already present) in the container as **/imagedata**.

```
% docker run --gpus all -ti --rm -v data:/imagedata nvcr.io/nvidia/
<repository>:<tag>
```

6.7. Setting The Mapping Ports Flag

Applications such as Deep Learning GPU Training System™ (DIGITS) open a port for communications. You can control whether that port is open only on the local system or is available to other computers on the network outside of the local system.

Using DIGITS as an example, in DIGITS 5.0 starting in container image 16.12, by default the DIGITS server is open on port 5000. However, after the container is started, you may not easily know the IP address of that container. To know the IP address of the container, you can choose one of the following ways:

- Expose the port using the local system network stack (`--net=host`) where port 5000 of the container is made available as port 5000 of the local system.

or

- Map the port (`-p 8080:5000`) where port 5000 of the container is made available as port 8080 of the local system.

In either case, users outside the local system have no visibility that DIGITS is running in a container. Without publishing the port, the port is still available from the host, however not from the outside.

6.8. Setting The Shared Memory Flag

Certain applications, such as PyTorch™ and the Microsoft® Cognitive Toolkit™ , use shared memory buffers to communicate between processes. Shared memory can also be required by single process applications, such as MXNet™ and TensorFlow™ , which use the NVIDIA® Collective Communications Library ™ (NCCL) (NCCL).

By default, Docker containers are allotted 64MB of shared memory. This can be insufficient, particularly when using all 8 GPUs. To increase the shared memory limit to a specified size, for example `1GB`, include the `--shm-size=1g` flag in your `docker run` command.

Alternatively, you can specify the `--ipc=host` flag to re-use the host's shared memory space inside the container. Though this latter approach has security implications as any data in shared memory buffers could be visible to other containers.

6.9. Setting The Restricting Exposure Of GPUs Flag

From inside the container, the scripts and software are written to take advantage of all available GPUs. To coordinate the usage of GPUs at a higher level, you can use this flag to restrict the exposure of GPUs from the host to the container. For example, if you only want GPU 0 and GPU 1 to be seen in the container, you would issue the following:

Using native GPU support

```
$ docker run --gpus "device=0,1" ...
```

Using `nvidia-docker2`

```
$ NV_GPU=0,1 docker run --runtime=nvidia ...
```

Using `nvidia-docker`

```
$ NV_GPU=0,1 nvidia-docker run ...
```

This flag creates a temporary environment variable that restricts which GPUs are used.

Specified GPUs are defined per container using the Docker device-mapping feature, which is currently based on Linux **`cgroups`**.

6.10. Container Lifetime

The state of an exited container is preserved indefinitely if you do not pass the **`--rm`** flag to the **`docker run`** command. You can list all of the saved exited containers and their size on the disk with the following command:

```
$ docker ps --all --size --filter Status=exited
```

The container size on the disk depends on the files created during the container execution, therefore the exited containers take only a small amount of disk space.

You can permanently remove a exited container by issuing:

```
docker rm [CONTAINER ID]
```

By saving the state of containers after they have exited, you can still interact with them using the standard Docker commands. For example:

- You can examine logs from a past execution by issuing the **`docker logs`** command.

  ```
  $ docker logs 9489d47a054e
  ```
- You can extract files using the **`docker cp`** command.

  ```
  $ docker cp 9489d47a054e:/log.txt .
  ```
- You can restart a stopped container using the **`docker restart`** command.

  ```
  $ docker restart <container name>
  ```

  For the NVCaffe™ container, issue this command:

  ```
  $ docker restart caffe
  ```
- You can save your changes by creating a new image using the **`docker commit`** command. For more information, see Example 3: Customizing a Container using .

> Use care when committing Docker container changes, as data files created during use of the container will be added to the resulting image. In particular, core dump files and logs can dramatically increase the size of the resulting image.

Chapter 7. RUNNING NGC CONTAINERS USING SINGULARITY

7.1. Overview

This chapter provides step-by-step instructions for pulling HPC containers from the NGC registry and running them using Singularity.

Singularity v2.6+ provides native Docker registry support. This support allows most Docker images, such as those hosted on NGC, to be pulled and converted to Singularity-compatible images in a single step.

NVIDIA tests HPC container compatibility with Singularity by converting the NGC images to Singularity format and running them through a rigorous QA process. The instructions below walk you through the process of pulling and running NGC containers with the Singularity runtime. Application-specific information may vary so it is recommended that you follow the container specific documentation before running with Singularity. If the container documentation does not include Singularity information, then the respective container was not tested using Singularity.

7.2. Prerequisites

These instructions assume the following.

- You have Singularity **v2.6+** installed on your system
- You have performed the following steps from the NGC website (see [NGC Getting Started Guide).
    - Signed up for an NGC account at https://ngc.nvidia.com/signup.
    - Created an NGC API key for access to the NGC container registry
    - Browsed the NGC website and identified an available NGC container and tag to run

- Ensure you have correctly set udev rules which are detailed (here)

It is recommended that you install **`nvidia-container-cli`** because if installed, Singularity will use it. More information can be found here.

7.3. Converting to Singularity Image

Before running with Singularity you must set NGC container registry authentication credentials.

This is most easily accomplished by setting the following environment variables.

bash

```
$ export SINGULARITY_DOCKER_USERNAME='$oauthtoken'
$ export SINGULARITY_DOCKER_PASSWORD=<NVIDIA NGC API key>
```

tcsh

```
> setenv SINGULARITY_DOCKER_USERNAME '$oauthtoken'
> setenv SINGULARITY_DOCKER_PASSWORD <NVIDIA NGC API key>
```

More information describing how to obtain and use your NVIDIA NGC API key can be found here.

Once credentials are set in the environment, the NGC container can be pulled to a local Singularity image.

```
$ singularity build <app_tag>.simg docker://nvcr.io/<repository>/<app:tag>
```

This will save the container to the current directory as

```
<app_tag>.simg
```

For example to convert the HPC application NAMD hosted on NGC to a Singularity image, run

```
$ singularity build namd_2.12-171025.simg docker://nvcr.io/hpc/namd:2.12-171025
```

After the build has finished the Singularity image file, namd_2.12-171025.simg, will be available for use in the current working directory.

7.4. Running the Singularity Container

Once the local Singularity image has been pulled, the following modes of running are supported.

- Command line execution with Singularity
- Interactive shell with Singularity

To leverage NVIDIA GPUs, you must use the Singularity flag `**--nv**`when running the containers. More singularity flags are explained here.

Important

For Amazon Machine Image Users

Amazon Machine Images on Amazon Web Service have a default root umask of 077. Singularity must be installed with a umask of 022 to run properly. To (re)install Singularity with correct permissions:

- Uninstall Singularity (if it is installed)
- Change the umask with: `` `$ umask 0022` ``
- Install Singularity
- Restore the umask: `` `$ umask 0077` ``

This causes installed Singularity files to have permission 0755 instead of the default 0700. Note that the umask command only applies changes to the current shell. Use umask and install Singularity from the same shell session.

7.4.1. Directory Access

Singularity containers are themselves ostensibly read only. In order to provide application input and persist application output we'll bind a host directory into our container, this is accomplished through the Singularity `` `-B` `` flag. The format of this flag is `` `-B <host_src_dir>:<container_dst_dir>` ``. Once a host directory is bound into the container we can interact with this directory from within the container exactly as we can outside the container.

It is also often convenient to use the `` `--pwd <container_dir>` `` flag, which will set the present working directory of the command to be run within the container.

The Singularity commands below will mount the present working directory on the host to `` `/host_pwd` `` in the container process and set the present working directory of the container process to `` `/host_pwd` ``. With this set of flags the `` `<cmd>` `` to be run will be launched from the host directory Singularity was called from.

```
$ singularity exec --nv -B $(pwd):/host_pwd --pwd /host_pwd <image.simg> <cmd>
```

Note: Binding to a directory which doesn't exist within the container image requires kernel and configuration support that may not be available on all systems, particularly those running older kernels such as CentOS/RHEL 6. When in doubt contact your system administrator.

7.4.2. Command Line Execution with Singularity

Running the container with Singularity from the command line looks similar to the command below.

```
$ singularity exec --nv <app_tag>.simg <command_to_run>
```

For example, to run the NAMD executable in the container

```
$ singularity exec --nv namd_2.12-171025.simg /opt/namd/namd-multicore
```

7.4.3. Interactive Shell with Singularity

To start a shell within the container, run the command below

```
$ singularity exec --nv <app_tag>.simg /bin/bash
```

For example, to start an interactive shell in the NAMD container

```
$ singularity exec --nv namd_2.12-171025.simg
```

Chapter 8. CUSTOMIZING CONTAINERS

NGC images come prepackaged, tuned, and ready to run; however, you may want to build a new image from scratch or augment an existing image with custom code, libraries, data, or settings for your corporate infrastructure. This section will guide you through exercises that will highlight how to create a container from scratch, customize a container, extend a deep learning framework to add features, develop some code using that extended framework from the developer environment, then package that code as a versioned release.

By default, you do not need to build a container. The NGC container registry, **`nvcr.io`**, has a number of containers that can be used immediately. These include containers for deep learning, scientific computing and visualization, as well as containers with just the CUDA Toolkit.

One of the great things about containers is that they can be used as starting points for creating new containers. This can be referred to as "customizing" or "extending" a container. You can create a container completely from scratch, however, since these containers are likely to run on a GPU system, it is recommended that you are least start with a **`nvcr.io`** container that contains the OS and CUDA. However, you are not limited to this and can create a container that runs on the CPUs in the system which does not use the GPUs. In this case, you can start with a bare OS container from Docker. However, to make development easier, you can still start with a container with CUDA - it is just not used when the container is used.

In the case of the DGX-1 and the DGX Station, you can push or save your modified/ extended containers to the NVIDIA DGX container registry, **`nvcr.io`**. They can also be shared with other users of the DGX system but this requires some administrator help.

Currently, you cannot save customized containers from the NGC container registry (cloud based) solution to **`nvcr.io`**. The customized or extended containers can be saved to a user's private container repository. The customized or extended containers can be saved to a user's private container repository.

It is important to note that all NGC deep learning framework images include the source to build the framework itself as well as all of the prerequisites.

Attention Do not install an NVIDIA driver into the Docker® image at Docker build time. nvidia-docker is essentially a wrapper around Docker that transparently provisions a container with the necessary components to execute code on the GPU.

NVIDIA provides a large set of images in the NGC container registry that are already tested, tuned, and are ready to run. You can pull any one of these images to create a container and add software or data of your choosing.

A best-practice is to avoid **`docker commit`** usage for developing new docker images, and to use Dockerfiles instead. The Dockerfile method provides visibility and capability to efficiently version-control changes made during development of a docker image. The docker commit method is appropriate for short-lived, disposable images only (see Example 3: Customizing A Container Using docker commit for an example).

For more information on writing a Docker file, see the best practices documentation.

8.1. Benefits And Limitations To Customizing A Container

You can customize a container to fit your specific needs for numerous reasons; for example, you depend upon specific software that is not included in the container that NVIDIA provides. No matter your reasons, you can customize a container.

The container images do not contain sample data-sets or sample model definitions unless they are included with the framework source. Be sure to check the container for sample data-sets or models.

8.2. Example 1: Building A Container From Scratch

Docker uses Dockerfiles to create or build a Docker image. Dockerfiles are scripts that contain commands that Docker uses successively to create a new Docker image. Simply put, a Dockerfile is the source code for the container image. Dockerfiles always start with a base image to inherit from.

For more information, see Best practices for writing Dockerfiles.

1. Create a working directory on your local hard-drive.
2. In that directory, open a text editor and create a file called **`Dockerfile`**. Save the file to your working directory.
3. Open your **`Dockerfile`** and include the following:

```
FROM ubuntu:14.04
RUN apt-get update && apt-get install -y curl
CMD echo "hello from inside a container"
```

Where the last line **`CMD`**, executes the indicated command when creating the container. This is a way to check that the container was built correctly.

For this example, we are also pulling the container from the Docker repository and not the DGX™ system repository. There will be subsequent examples using the NVIDIA® repository.

4. Save and close your **`Dockerfile`**.
5. Build the image. Issue the following command to build the image and create a tag.

```
$ docker build -t <new_image_name>:<new_tag> .
```

This command was issued in the same directory where the Dockerfile is located.

The output from the docker build process lists “Steps”; one for each line in the Dockerfile.

For example, let us name the container **`test1`** and tag it with **`latest`**. Also, for illustrative purposes, let us assume our private DGX system repository is called **`nvidian_sas`**. The command below builds the container. Some of the output is shown below so you know what to expect.

```
$ docker build -t test1:latest .
Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM ubuntu:14.04
14.04: Pulling from library/ubuntu
...
Step 2/3 : RUN apt-get update && apt-get install -y curl
...
Step 3/3 : CMD echo "hello from inside a container"
 ---> Running in 1f491b9235d8
 ---> 934785072daf
Removing intermediate container 1f491b9235d8
Successfully built 934785072daf
```

For information about building your image, see docker build. For information about tagging your image, see docker tag.

6. Verify that the build was successful. You should see a message similar to the following:

```
Successfully built  934785072daf
```

This message indicates that the build was successful. Any other message and the build was not successful.

The number, **`934785072daf`**, is assigned when the image is built and is random.

7. Confirm you can view your image. Issue the following command and view your container.

```
$ docker images
REPOSITORY          TAG             IMAGE ID        CREATED                 SIZE
test1               latest          934785072daf    19 minutes ago          222 MB
```

The new container is now available to be used.

The container is local to this DGX system. If you want to store the container in your private repository, follow the next step.

8. Store the container in your private Docker repository by pushing it.

   This only works for the DGX-1™ and the DGX Station.

   a) The first step in pushing it, is to tag it.

   ```
   $ docker tag test1 nvcr.io/nvidian_sas/test1:latest
   ```

   b) Now that the image has been tagged, you can push it to, for example, a private project on **nvcr.io** named **nvidian_sas**.

   ```
   $ docker push nvcr.io/nvidian_sas/test1:latest
   The push refers to a repository [nvcr.io/nvidian_sas/test1]
   …
   ```

   c) Verify that the container appears in the **nvidian_sas** repository.

8.3. Example 2: Customizing A Container Using Dockerfile

This example uses a Dockerfile to customize the NVCaffe container in **nvcr.io**. Before customizing the container, you should ensure the NVCaffe 17.03 container has been loaded into the registry using the **docker pull** command before proceeding.

```
$ docker pull nvcr.io/nvidia/caffe:17.03
```

As mentioned earlier in this document, the Docker containers on **nvcr.io** also provide a sample Dockerfile that explains how to patch a framework and rebuild the Docker image. In the directory **/workspace/docker-examples**, there are two sample Dockerfiles. For this example, we will use the **Dockerfile.customcaffe** file as a template for customizing a container.

1. Create a working directory called **my_docker_images** on your local hard drive.
2. Open a text editor and create a file called **Dockerfile**. Save the file to your working directory.
3. Open your **Dockerfile** again and include the following lines in the file:

```
FROM nvcr.io/nvidia/caffe:17.03
# APPLY CUSTOMER PATCHES TO CAFFE
# Bring in changes from outside container to /tmp
# (assumes my-caffe-modifications.patch is in same directory as
Dockerfile)
#COPY my-caffe-modifications.patch /tmp

# Change working directory to NVCaffe source path
WORKDIR /opt/caffe
```

```
# Apply modifications
#RUN patch -p1 < /tmp/my-caffe-modifications.patch

# Note that the default workspace for caffe is /workspace
RUN mkdir build && cd build && \
  cmake -DCMAKE_INSTALL_PREFIX:PATH=/usr/local -DUSE_NCCL=ON
-DUSE_CUDNN=ON -DCUDA_ARCH_NAME=Manual -DCUDA_ARCH_BIN="35 52 60 61"
-DCUDA_ARCH_PTX="61" .. && \
  make -j"$(nproc)" install && \
  make clean && \
  cd .. && rm -rf build

# Reset default working directory
WORKDIR /workspace
```

Save the file.

4. Build the image using the **`docker build`** command and specify the repository name and tag. In the following example, the repository name is *corp/caffe* and the tag is *17.03.1PlusChanges*. For the case, the command would be the following:

```
$ docker build -t corp/caffe:17.03.1PlusChanges .
```

5. Run the Docker image using the command appropriate to the methoid of GPU support installed.

```
$ docker run --gpus all -ti --rm corp/caffe:17.03.1PlusChanges .
```

The base command **`docker run --gpu all`** assumes that your system has Docker 19.03-CE installed. See the section Enabling GPU Support for NGC Containers for the command to use for earlier versions of Docker.

8.4. Example 3: Customizing A Container Using `docker commit`

This example uses the `docker commit` command to flush the current state of the container to a Docker image. This is not a recommended best practice, however, this is useful when you have a container running to which you have made changes and want to save them. In this example, we are using the **`apt-get`** tag to install packages which requires that the user run as root.

- The NVCaffe image release 17.04 is used in the example instructions for illustrative purposes.
- Do not use the **`--rm`** flag when running the container. If you use the **`--rm`** flag when running the container, your changes will be lost when exiting the container.

1. Pull the Docker container from the **`nvcr.io`** repository to the DGX system. For example, the following command will pull the NVCaffe container:

```
$ docker pull nvcr.io/nvidia/caffe:17.04
```

2. Run the container on the DGX system.

```
$ docker run --gpus all -ti nvcr.io/nvidia/caffe:17.04
```

```
==================
== NVIDIA Caffe ==
==================

NVIDIA Release 17.04 (build 26740)

Container image Copyright (c) 2017, NVIDIA CORPORATION.  All rights
 reserved.
Copyright (c) 2014, 2015, The Regents of the University of California
 (Regents)
All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION.  All rights
 reserved.
NVIDIA modifications are covered by the license terms that apply to the
 underlying project or file.

NOTE: The SHMEM allocation limit is set to the default of 64MB.  This may be
 insufficient for NVIDIA Caffe.  NVIDIA recommends the use of the following
 flags:
   nvidia-docker run --shm-size=1g --ulimit memlock=-1 --ulimit
 stack=67108864 ...

root@1fe228556a97:/workspace#
```

The base command **`docker run --gpu all`** assumes that your system has Docker 19.03-CE installed. See the section Enabling GPU Support for NGC Containers for the command to use for earlier versions of Docker.

3. You should now be the root user in the container (notice the prompt). You can use the command **`apt`** to pull down a package and put it in the container.

   The NVIDIA containers are built using Ubuntu which uses the **`apt-get`** package manager. Check the container release notes Deep Learning Documentation for details on the specific container you are using.

   In this example, we will install Octave; the GNU clone of MATLAB, into the container.

```
# apt-get update
# apt install octave
```

   You have to first issue **`apt-get update`** before you install Octave using **`apt`**.

4. Exit the workspace.

```
# exit
```

5. Display the list of containers using **`docker ps -a`**. As an example, here is some of the output from the **`docker ps -a`** command:

```
$ docker ps -a
CONTAINER ID    IMAGE                         CREATED        ...
1fe228556a97    nvcr.io/nvidia/caffe:17.04    3 minutes ago  ...
```

6. Now you can create a new image from the container that is running where you have installed Octave. You can commit the container with the following command.

```
$ docker commit 1fe228556a97 nvcr.io/nvidian_sas/caffe_octave:17.04
sha256:0248470f46e22af7e6cd90b65fdee6b4c6362d08779a0bc84f45de53a6ce9294
```

7. Display the list of images.

```
$ docker images
REPOSITORY                    TAG                IMAGE ID     ...
nvidian_sas/caffe_octave      17.04              75211f8ec225 ...
```

8. To verify, let's run the container again and see if Octave is actually there.

This only works for the DGX-1 and the DGX Station.

```
$ docker run --gpus all -ti nvidian_sas/caffe_octave:17.04
```

```
==================
== NVIDIA Caffe ==
==================

NVIDIA Release 17.04 (build 26740)

Container image Copyright (c) 2017, NVIDIA CORPORATION.  All rights
 reserved. Copyright (c) 2014, 2015, The Regents of the University of
 California (Regents) All rights reserved.

Various files include modifications (c) NVIDIA CORPORATION.  All rights
 reserved. NVIDIA modifications are covered by the license terms that apply
 to the underlying project or file.

NOTE: The SHMEM allocation limit is set to the default of 64MB.  This may be
 insufficient for NVIDIA Caffe.  NVIDIA recommends the use of the following
 flags:
   nvidia-docker run --shm-size=1g --ulimit memlock=-1 --ulimit
 stack=67108864 ...

root@2fc3608ad9d8:/workspace# octave
octave: X11 DISPLAY environment variable not set
octave: disabling GUI features
GNU Octave, version 4.0.0
Copyright (C) 2015 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  For details, type 'warranty'.

Octave was configured for "x86_64-pc-linux-gnu".

Additional information about Octave is available at http://www.octave.org.

Please contribute if you find this software useful.
For more information, visit http://www.octave.org/get-involved.html

Read http://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1>
```

Since the Octave prompt displayed, Octave is installed.

9. If you want to save the container into your private repository (Docker uses the phrase “push”), then you can use the command **`docker push ...`**.

```
$ docker push nvcr.io/nvidian_sas/caffe_octave:17.04
```

The new Docker image is now available for use. You can check your local Docker repository for it.

8.5. Example 4: Developing A Container Using Docker

There are two primary use cases for a developer to extend a container:

1. Create a development image that contains all of the immutable dependencies for the project, but not the source code itself.
2. Create a production or testing image that contains a fixed version of the source and all of the software dependencies.

The datasets are not packaged in the container image. Ideally, the container image is designed to expect volume mounts for datasets and results.

In these examples, we mount our local dataset from **`/raid/datasets`** on our host to **`/dataset`** as a read-only volume inside the container. We also mount a job specific directory to capture the output from a current run.

In these examples, we will create a timestamped output directory on each container launch and map that into the container at **`/output`**. Using this method, the output for each successive container launch is captured and isolated.

Including the source into a container for developing and iterating on a model has many awkward challenges that can over complicate the entire workflow. For instance, if your source code is in the container, then your editor, version control software, dotfiles, etc. also need to be in the container.

However, if you create a development image that contains everything you need to run your source code, you can map your source code into the container to make use of your host workstation's developer environment. For sharing a fixed version of a model, it is best to package a versioned copy of the source code and trained weights with the development environment.

As an example, we will work though a development and delivery example for the open source implementation of the work found in Image-to-Image Translation with Conditional Adversarial Networks by Isola et. al. and is available at pix2pix. Pix2Pix is a Torch implementation for learning a mapping from input images to output images using a Conditional Adversarial Network. Since online projects can change over time, we will focus our attention on the snapshot version **`d7e7b8b557229e75140cbe42b7f5dbf85a67d097`** change-set.

In this section, we are using the container as a virtual environment, in that the container has all the programs and libraries needed for our project.

We have kept the network definition and training script separate from the container image. This is a useful model for iterative development because the files that are actively being worked on are persistent on the host and only mapped into the container at runtime.

The differences to the original project can be found here Comparing changes.

If the machine you are developing on is not the same machine on which you will be running long training sessions, then you may want to package your current development state in the container.

1. Create a working directory on your local hard-drive.

```
mkdir Projects
$ cd ~/Projects
```

2. Git clone the Pix2Pix git repository.

```
$ git clone https://github.com/phillipi/pix2pix.git
$ cd pix2pix
```

3. Run the `git checkout` command.

```
$ git checkout -b devel d7e7b8b557229e75140cbe42b7f5dbf85a67d097
```

4. Download the dataset:

```
bash ./datasets/download_dataset.sh facades

I want to put the dataset on my fast /raid storage.
$ mkdir -p /raid/datasets
$ mv ./datasets/facades /raid/datasets
```

5. Create a file called **`Dockerfile`**, and add the following lines:

```
FROM nvcr.io/nvidia/torch:17.03
RUN luarocks install nngraph
RUN luarocks install
https://raw.githubusercontent.com/szym/display/master/display-scm-0.rockspec
WORKDIR /source
```

6. Build the development Docker container image (**`build-devel.sh`**).

```
docker build -t nv/pix2pix-torch:devel .
```

7. Create the following **`train.sh`** script:

```
#!/bin/bash -x
ROOT="${ROOT:-/source}"
DATASET="${DATASET:-facades}"
DATA_ROOT="${DATA_ROOT:-/datasets/$DATASET}"
DATA_ROOT=$DATA_ROOT name="${DATASET}_generation"
which_direction=BtoA th train.lua
```

   If you were actually developing this model, you would be iterating by making changes to the files on the host and running the training script which executes inside the container.

8. Optional: Edit the files and execute the next step after each change.
9. Run the training script (**`run-devel.sh`**).

```
docker run --gpus all --rm -ti -v $PWD:/source  -v
/raid/datasets:/datasets nv/pix2pix-torch:devel ./train.sh
```

> The base command **`docker run --gpu all`** assumes that your system has Docker 19.03-CE installed. See the section Enabling GPU Support for NGC Containers for the command to use for earlier versions of Docker.

8.5.1. Example 4.1: Package The Source Into The Container

Packaging the model definition and script into the container is very simple. We simply add a **`COPY`** step to the Dockerfile.

We've updated the run script to simply drop the volume mounting and use the source packaged in the container. The packaged container is now much more portable than our **`devel`** container image because the internal code is fixed. It would be good practice to version control this container image with a specific tag and store it in a container registry.

The updates to run the container are equally subtle. We simply drop the volume mounting of our local source into the container.

Chapter 9. TROUBLESHOOTING

For more information about nvidia-docker containers, visit the GitHub site: NVIDIA-Docker GitHub.

For deep learning frameworks release notes and additional product documentation, see the Deep Learning Documentation website: Release Notes for Deep Learning Frameworks.