NVIDIA Containers For Deep Learning Frameworks

User Guide

DU-08518-001_v001 | April 2024

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.

The NVIDIA Container Runtime for Docker, also known as nvidia-docker2 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. It accomplishes this through the use of Docker containers.

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

Docker container
A Docker container is an instance of a Docker image. A Docker container deploys a single application or service per 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 composed of layers. The layers are combined to create the container. You can think of layers as intermediate images that add some capability to the overall container. If you make a change to a layer through a DockerFile (see Building Containers), than Docker rebuilds that layer and all subsequent layers but not the layers that are not affected by the build. This reduces the time to create containers and also allows you to keep them modular.

Docker is also very good about keeping one copy of the layers on a system. This saves space and also greatly reduces the possibility of “version skew” so that layers that should be the same are not duplicated.

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.

1.3. Hello World For Containers

To make sure you have access to the NVIDIA containers, start with the proverbial “hello world” of Docker commands.

For DGX systems, simply log into the system. See the Frameworks Support Matrix for the current list of DGX systems supported. For NGC consult the NGC documentation for details about your specific cloud provider. In general, you will start a cloud instance with your cloud provider using the NVIDIA Volta Deep Learning Image. After the instance has booted, log into the instance.

Next, you can issue the `docker --version` command to list the version of DGX systems. The output of this command tells you the version of Docker on the system (`18.06.3-ce, build 89658be`).

At any time, if you are not sure about a Docker command, issue the `docker --help` command.

1.4. Logging Into Docker

If you have a DGX system, the first time you login, you are required to set-up access to the NVIDIA NGC container registry (https://ngc.nvidia.com). For more information, see the NGC Getting Started Guide.

1.5. Listing Docker Images

Typically, one of the first things you will want to do is get a list of all the Docker images that are currently available on the local computer. Issuing a `docker pull` command will download Docker images from the repository onto your local system.

Issue the `docker images` command to list the images on the server. Your screen will look similar to the following:

```
REPOSITORY                              TAG                             IMAGE ID
mxnet-dec                               latest                          65a48ellda96
<none>                                  <none>                          bfc4512ca5f2
nvcr.io/nvidian_general/adlr_pytorch    resumes                         al34a09668a8
<none>                                  <none>                          0f4ab6d62241
<none>                                  <none>                          97274da5c898
nvcr.io/nvidian_sas/games-libcchem      cuda10                          3dcl3f8347f9
nvidia/cuda                             latest                          614dcdafa05c
ubuntu                                  latest                          d355ed3537e9
deeper_photo                            latest                          932634514d5a
nvcr.io/nvidia/caffe                    19.03                           b7b62dacdebl
nvidia/cuda                             10.0-devel-centos7              6e3e5b71176e
nvcr.io/nvidia/tensorflow               19.03                           56f2980ble37
nvidia/cuda                             10.0-cudnn7-devel-ubuntu16.04   22afb0578249
nvidia/cuda                             10.0-devel                      a760a0cfca82
mxnet/python                            gpu                             7e7c9176319c
nvcr.io/nvidia_sas/chainer              latest                          2ea707c58bea
deep_photo                              latest                          ef4510510506
<none>                                  <none>                          9124236672fe
nvcr.io/nvidia/cuda                     10.0-cudnn7-devel-ubuntu18.04   02910409eb5d
nvcr.io/nvidia/tensorflow               19.05                           9dda0d5c344f
```

In this example, there are a few Docker containers that have been pulled down to this system. Each image is listed along with its *tag*, the corresponding *Image ID*, also known as *container version*. There are two other columns that list when the container was created (approximately), and the approximate size of the image in GB. These columns have been cropped to improve readability.

Note: The output from the command will vary.

At any time, if you need help, issue the `docker images --help` command.

Chapter 2. Installing Docker And NVIDIA Container Runtime

About this task

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

- Native GPU support
- nvidia-docker2

Each of these methods provide a command line tool to mount the user-mode components of the NVIDIA driver and the GPUs into the Docker container at launch.

NVIDIA has also developed a set of containers which includes software that is specific to NVIDIA DGX systems. These containers ensure the best performance for your applications and should provide the best single-GPU performance and multi-GPU scaling.

NGC containers take full advantage of NVIDIA GPUs. HPC visualization containers have differing prerequisites than DGX containers. For more information, see the NGC Container User Guide.

For installation instructions, refer to the Quick Start section in the NVIDIA Container Toolkit GitHub repository.

2.1. Docker Best Practices

You can run a Docker container on any platform that is Docker compatible allowing you to move your application to wherever you need. The containers are platform-agnostic, and therefore, hardware agnostic as well. To get the best performance and to take full advantage of the tremendous performance of an NVIDIA GPU, specific kernel modules and user-level libraries are needed. NVIDIA GPUs introduce some complexity because they require kernel modules and user-level libraries to operate.

One approach to solving this complexity when using containers is to have the NVIDIA drivers installed in the container and have the character devices mapped corresponding to the NVIDIA GPUs such as `/dev/nvidia0`. For this to work, the drivers on the host (the

system that is running the container), must match the version of the driver installed in the container. This approach drastically reduces the portability of the container.

2.2. docker exec

There are times when you will need to connect to a running container. You can use the `docker exec` command to connect to a running container to run commands. You can use the `bash` command to start an interactive command line terminal or bash shell.

```
$ docker exec -it <CONTAINER_ID_OR_NAME> bash
```

As an example, suppose one starts a Deep Learning GPU Training System™ (DIGITS) container with the following command:

```
docker run --gpus all -d --name test-digits \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/digits:17.05
```

After the container is running, you can now connect to the container instance.

```
$ docker exec -it test-digits bash
```

Note: `test-digits` is the name of the container. If you don't specifically name the container, you will have to use the container ID.

Important: By using `docker exec`, you can execute a snippet of code, a script, or attach interactively to the container making the `docker exec` command very useful.

For detailed usage of the `docker exec` command, see docker exec.

2.3. nvcr.io

Building deep learning frameworks can be quite a bit of work and can be very time consuming. Moreover, these frameworks are being updated weekly, if not daily. On top of this, is the need to optimize and tune the frameworks for GPUs. NVIDIA has created a Docker repository, named `nvcr.io`, where deep learning frameworks are tuned, optimized, tested, and containerized for your use.

NVIDIA creates an updated set of Docker containers for the frameworks monthly. Included in the container is source (these are open-source frameworks), scripts for building the frameworks, Dockerfiles for creating containers based on these containers, markdown files that contain text about the specific container, and tools and scripts for pulling down datasets that can be used for testing or learning. Customers who purchase a DGX system have access to this repository for pushing containers (storing containers).

To get started with DGX systems, you need to create a system admin account for accessing `nvcr.io`. This account should be treated as an admin account so that users cannot access it. Once this account is created, the system admin can create accounts for projects that belong to the account. They can then give users access to these projects so that they can store or share any containers that they create.

2.4. Building Containers

About this task

You can build and store containers in the `nvcr.io` registry as a project within your account if you have a DGX system (for example, no one else can access the container unless you give them access).

This section of the document applies to Docker containers in general. You can use this general approach for your own Docker repository as well, but be cautious of the details.

Using a DGX system you can either:

1. Create your container from scratch
2. Base your container on an existing Docker container
3. Base your container on containers in `nvcr.io`.

Any one of the three approaches are valid and will work, however, since the goal is to run the containers on a system which has GPUs, it's logical to assume that the applications will be using GPUs. Moreover, these containers are already tuned for GPUs. All of them also include the needed GPU libraries, configuration files, and tools to rebuild the container.

Important: Based on these assumptions, it's recommended that you start with a container from `nvcr.io`.

An existing container in `nvcr.io` should be used as a starting point. As an example, the TensorFlow 17.06 container will be used and Octave will be added to the container so that some post-processing of the results can be accomplished.

Procedure

1. Pull the container from the NGC container registry to the server. See Pulling A Container.
2. On the server, create a subdirectory called `mydocker`.

   Note: This is an arbitrary directory name.

3. Inside this directory, create a file called `Dockerfile` (capitalization is important). This is the default name that Docker looks for when creating a container. The `Dockerfile` should look similar to the following:

```
[username ~]$ mkdir mydocker
[username ~]$ cd mydocker
[username mydocker]$ vi Dockerfile
```

```
[username mydocker]$ more Dockerfile
FROM nvcr.io/nvidia/tensorflow:19.03

RUN apt-get update

RUN apt-get install -y octave
[username mydocker]$
```

There are three lines in the `Dockerfile`.

- The first line in the `Dockerfile` tells Docker to start with the container `nvcr.io/nvidia/tensorflow:17.06`. This is the base container for the new container.
- The second line in the `Dockerfile` performs a package update for the container. It doesn't update any of the applications in the container, but updates the `apt-get` database. This is needed before we install new applications in the container.
- The third and last line in the `Dockerfile` tells Docker to install the package `octave` into the container using `apt-get`.

The Docker command to create the container is:

```
$ docker build -t nvcr.io/nvidian_sas/tensorflow_octave:17.06_with_octave
```

Note: This command uses the default file `Dockerfile` for creating the container.

The command starts with `docker build`. The `-t` option creates a tag for this new container. Notice that the tag specifies the project in the `nvcr.io` repository where the container is to be stored. As an example, the project `nvidian_sas` was used along with the repository `nvcr.io`.

Projects can be created by your local administrator who controls access to `nvcr.io`, or they can give you permission to create them. This is where you can store your specific containers and even share them with your colleagues.

```
[username mydocker]$ docker build -t nvcr.io/nvidian_sas/
tensorflow_octave:19.03_with_octave .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM nvcr.io/nvidia/tensorflow:1903
 ---> 56f2980b1e37
Step 2/3 : RUN apt-get update
 ---> Running in 69cffa7bbadd
Get:1 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Get:2 http://ppa.launchpad.net/openjdk-r/ppa/ubuntu xenial InRelease [17.5 kB]
Get:3 http://archive.ubuntu.com/ubuntu xenial InRelease [247 kB]
Get:4 http://ppa.launchpad.net/openjdk-r/ppa/ubuntu xenial/main amd64 Packages [7096 B]
Get:5 http://security.ubuntu.com/ubuntu xenial-security/universe Sources [42.0 kB]
Get:6 http://security.ubuntu.com/ubuntu xenial-security/main amd64 Packages [380 kB]
Get:7 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Get:8 http://security.ubuntu.com/ubuntu xenial-security/restricted amd64 Packages [12.8
 kB]
Get:9 http://security.ubuntu.com/ubuntu xenial-security/universe amd64 Packages [178 kB]
Get:10 http://security.ubuntu.com/ubuntu xenial-security/multiverse amd64 Packages [2931
 B]
Get:11 http://archive.ubuntu.com/ubuntu xenial-backports InRelease [102 kB]
Get:12 http://archive.ubuntu.com/ubuntu xenial/universe Sources [9802 kB]
Get:13 http://archive.ubuntu.com/ubuntu xenial/main amd64 Packages [1558 kB]
Get:14 http://archive.ubuntu.com/ubuntu xenial/restricted amd64 Packages [14.1 kB]
Get:15 http://archive.ubuntu.com/ubuntu xenial/universe amd64 Packages [9827 kB]
```

In the brief output from the `docker build …` command shown above, each line in the Dockerfile is a *Step*. In the screen capture, you can see the first and second steps

(commands). Docker echos these commands to the standard out (`stdout`) so you can watch what it is doing or you can capture the output for documentation.

After the image is built, remember that we haven't stored the image in a repository yet, therefore, it's a `docker image`. Docker prints out the image id to `stdout` at the very end. It also tells you if you have successfully created and tagged the image.

If you don't see `Successfully ...` at the end of the output, examine your `Dockerfile` for errors (perhaps try to simplify it) or try a very simple `Dockerfile` to ensure that Docker is working properly.

4. Verify that Docker successfully created the image.

```
$ docker images
```

For example:

```
[username mydocker]$ docker images
REPOSITORY                                TAG                                 IMAGE ID
 CREATED
nvcr.io/nvidian_sas/tensorflow_octave     19.03_with_octave                   67c448c6fe37  About
 a minute ago
nvcr.io/nvidian_general/adlr_pytorch      resumes                             17f2398a629e  47
 hours ago
<none>                                    <none>                              0c0f174e3bbc  9
 days ago
nvcr.io/nvidian_sas/pushed-hshin          latest                              c026c5260844  9
 days ago
torch-caffe                               latest                              a5cdc9173d02  11
 days ago
<none>                                    <none>                              a134a09668a8  2
 weeks ago
<none>                                    <none>                              0f4ab6d62241  2
 weeks ago
nvidia/cuda                               10.0-cudnn7-devel-ubuntu16.04       a995cebf5782  2
 weeks ago
mxnet-dec-abcd                            latest                              8bceaf5e58de  2
 weeks ago
keras_ae                                  latest                              92ab2bed8348  3
 weeks ago
nvidia/cuda                               latest                              614dcdafa05c  3
 weeks ago
ubuntu                                    latest                              d355ed3537e9  3
 weeks ago
deeper_photo                              latest                              f4e395972368  4
 weeks ago
<none>                                    <none>                              0e8208a5e440  4
 weeks ago
nvcr.io/nvidia/digits                     19.03                               c4e87f2a1ebe  5
 weeks ago
nvcr.io/nvidia/tensorflow                 19.03                               56f2980b1e37  5
 weeks ago
mxnet/python                              gpu                                 7e7c9176319c  6
 weeks ago
nvcr.io/nvidian_sas/chainer               latest                              2ea707c58bea  6
 weeks ago
deep_photo                                latest                              ef4510510506  7
 weeks ago
<none>                                    <none>                              9124236672fe  8
 weeks ago
nvcr.io/nvidia/cuda                       10.0-cudnn7-devel-ubuntu18.04       02910409eb5d  8
 weeks ago
nvcr.io/nvidia/digits                     19.03                               c14438dc0277  2
 months ago
nvcr.io/nvidia/tensorflow                 19.03                               9dda0d5c344f  2
 months ago
```

```
nvcr.io/nvidia/caffe                    19.03                               87c288427f2d  2
 months ago
nvcr.io/nvidia/tensorflow               19.03                               121558cb5849  3
 months ago
```

The very first entry is the new image (about 1 minute old).

5. Push the image into the repository, creating a container.

```
docker push <name of image>
```

For example:

```
[username mydocker]$ docker push nvcr.io/nvidian_sas/tensorflow_octave:19.03_with_octave
The push refers to a repository [nvcr.io/nvidian_sas/tensorflow_octave]
1b81f494d27d: Image successfully pushed
023cdba2c5b6: Image successfully pushed
8dd41145979c: Image successfully pushed
7cb16b9b8d56: Image already pushed, skipping
bd5775db0720: Image already pushed, skipping
bc0c86a33aa4: Image already pushed, skipping
cc73913099f7: Image already pushed, skipping
d49f214775fb: Image already pushed, skipping
5d6703088aa0: Image already pushed, skipping
7822424b3bee: Image already pushed, skipping
e999e9a30273: Image already pushed, skipping
e33eae9b4a84: Image already pushed, skipping
4a2ad165539f: Image already pushed, skipping
7efc092a9b04: Image already pushed, skipping
914009c26729: Image already pushed, skipping
4a7ea614f0c0: Image already pushed, skipping
550043e76f4a: Image already pushed, skipping
9327bc01581d: Image already pushed, skipping
6ceab726bc9c: Image already pushed, skipping
362a53cd605a: Image already pushed, skipping
4b74ed8a0e09: Image already pushed, skipping
1f926986fb96: Image already pushed, skipping
832ac06c43e0: Image already pushed, skipping
4c3abd56389f: Image already pushed, skipping
d8b353eb3025: Image already pushed, skipping
f2e85bc0b7b1: Image already pushed, skipping
fc9ele5e38f7: Image already pushed, skipping
f39a3f9c4559: Image already pushed, skipping
6a8bf8c8edbd: Image already pushed, skipping
Pushing tag for rev [67c448c6fe37] on {https://nvcro.io/v1/repositories/nvidian_sas/
tensorflow_octave
```

The above sample code is after the `docker push …` command pushes the image to the repository creating a container. At this point, you should log into the NGC container registry at https://ngc.nvidia.com and look under your project to see if the container is there.

If you don't see the container in your project, make sure that the tag on the image matches the location in the repository. If, for some reason, the push fails, try it again in case there was a communication issue between your system and the container registry (`nvcr.io`).

To make sure that the container is in the repository, we can pull it to the server and run it. As a test, first remove the image from your DGX system using the command `docker rmi ….` Then, pull the container down to the server using `docker pull ….` Notice that the `octave` prompt came up so it is installed and functioning correctly within the limits of this testing.

2.5. Using And Mounting File Systems

One of the fundamental aspects of using Docker is mounting file systems inside the Docker container. These file systems can contain input data for the frameworks or even code to run in the container.

Docker containers have their own internal file system that is separate from file systems on the rest of the host.

Important: You can copy data into the container file system from outside if you want. However, it's far easier to mount an outside file system into the container.

Mounting outside file systems is done using the `docker run --gpus` command and the `-v` option. For example, the following command mounts two file systems:

```
$ docker run --gpus all --rm -ti ... -v $HOME:$HOME \
  -v /datasets:/digits_data:ro \
  ...
```

Most of the command has been erased except for the volumes. This command mounts the user's home directory from the external file system to the home directory in the container (`-v $HOME:$HOME`). It also takes the `/datasets` directory from the host and mounts it on `/digits_data` inside the container (`-v /datasets:/digits_data:ro`).

Remember: The user has root privileges with Docker, therefore you can mount almost anything from the host system to anywhere in the container.

For this particular command, the volume command takes the form of:

```
-v <External FS Path>:<Container FS Path>(options) \
```

The first part of the option is the path for the external file system. To be sure this works correctly, it's best to use the fully qualified path (FQP). This is also true for the mount point inside the container `<Container FS Path>`.

After the last path, various options can be used in parenthesis `()`. In the above example, the second file system is mounted read-only (`ro`) inside the container. The various options for the `-v` option are discussed here.

The DGX™ systems and the Docker containers use the Overlay2 storage driver to mount external file systems onto the container file system. Overlay2 is a union-mount file system driver that allows you to combine multiple file systems so that all the content appears to be combined into a single file system. It creates a *union* of the file systems rather than an intersection.

Chapter 3. Pulling A Container

About this task

Before you can pull a container from the NGC container registry, you must have Docker installed. For DGX users, this is explained in Preparing to use NVIDIA Containers Getting Started Guide.

For users other than DGX, follow the NVIDIA® GPU Cloud™ (NGC) container registry installation documentation based on your platform.

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

There are four repositories where you can find the NGC docker containers.

nvcr.io/nvidia
The deep learning framework containers are stored in the `nvcr.io/nvidia` repository.

nvcr.io/hpc
The HPC containers are stored in the `nvcr.io/hpc` repository.

nvcr.io/nvidia-hpcvis
The HPC visualization containers are stored in the `nvcr.io/nvidia-hpcvis` repository.

nvcr.io/partner
The partner containers are stored in the `nvcr.io/partner` repository. Currently the partner containers are focused on Deep Learning or Machine Learning, but that doesn't mean they are limited to those types of containers.

3.1. Key Concepts

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/tensorflow:<xx.xx>
```

Note: The base command `docker run --gpu all` assumes that your system has Docker 19.03-CE and the NVIDIA runtime packages 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 is `nvcr.io`.

nvidia
The name of the space within the registry that contains the deep learning container. For containers provided by NVIDIA, the registry space is `nvidia`.

-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 container version. For example, `19.01`.

py<x>
The Python version. For example, `py3`.

3.2. Accessing And Pulling From The NGC container registry

Before you begin

Before accessing the NGC container registry, ensure that the following prerequisites are met. For more information about meeting these requirements, see NGC Getting Started Guide.

- Create an account on the NGC container registry: https://ngc.nvidia.com. Ensure you store the API key somewhere safe since you will need it later. After you create an account, the commands to pull containers are the same as if you had a DGX system in your own data center.

  Note: You can access the NGC container registry by running a Docker command from your client computer. You are not limited to using your DGX platform to access the NGC container registry. You can use any Linux computer with Internet access on which Docker is installed. For more information about which platforms are supported, see https://docs.nvidia.com/ngc/index.html.

- Your NGC account is activated.
- You have an NGC API key for authenticating your access to the NGC container registry.
- You are logged in to your client computer with the privileges required to run Docker containers.

After your NGC account is activated, you can access the NGC container registry from one of two ways:

- Pulling A Container From The NGC container registry Using The Docker CLI
- Pulling A Container Using The NGC Web Interface

About this task

A Docker registry is the service that stores Docker images. The service can be on the internet, on the company intranet, or on a local machine. For example, `nvcr.io` is the location of the NGC container registry for Docker images.

All `nvcr.io` Docker images use explicit container-version-tags to avoid tagging issues which can result from using the latest tag. For example, a locally tagged “latest” version of an image may actually override a different “latest” version in the registry.

Procedure

1. Log in to the NGC container registry.
   ```
   $ docker login nvcr.io
   ```
2. When prompted for your username, enter the following text:
   ```
   $oauthtoken
   ```
   The `$oauthtoken` username is a special user name that indicates that you will authenticate with an API key and not a username and password.
3. When prompted for your password, enter your NGC API key.
   ```
   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. Also, be sure to store it somewhere safe because it's possible you may need it later.

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

Before you begin

Before pulling a container, ensure that the following prerequisites are met:

- You have read access to the registry space that contains the container.
- You are logged into the NGC container registry as explained in Accessing And Pulling From The NGC container registry and you have your API key stored somewhere safe that is also accessible.
- Your account is a member of the `docker` group, which enables you to use Docker commands.

Tip: To browse the available containers in the NGC container registry use a web browser to log into your NGC container registry account on the NGC website.

Procedure

1. Pull the container that you want from the registry. For example, to pull the PyTorch™ 21.02 container:
   ```
   $ docker pull nvcr.io/nvidia/pytorch:21.02-py3
   ```
2. List the Docker images on your system to confirm that the container was pulled.
   ```
   $ docker images
   ```

What to do next

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.

3.2.2. Pulling A Container Using The NGC Web Interface

Before you begin

Before you can pull a container from the NGC container registry, you must have Docker and nvidia-docker2 installed as explained in Preparing To Use NVIDIA Containers Getting

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

About this task

This task assumes:

1. You have a cloud instance system and it is connected to the Internet.
2. Your instance has Docker and nvidia-docker2 installed.
3. You have access to a browser to the NGC container registry at https://ngc.nvidia.com and your NGC account is activated.
4. You want to pull a container onto your cloud instance.

Procedure

1. Log into the NGC container registry at https://ngc.nvidia.com.
2. Click **Registry** in the left navigation. Browse the NGC container registry page to determine which Docker repositories and tags are available to you.
3. Click one of the repositories to view information about that container image as well as the available tags that you will use when running the container.
4. In the **Pull** column, click the icon to copy the Docker pull command.
5. Open a command prompt and paste the Docker pull command. The pulling of the container image begins. Ensure the pull completes successfully.
6. After you have the Docker container file on your local system, load the container into your local Docker registry.
7. Verify that the image is loaded into your local Docker registry.

   ```
   $ docker images
   ```

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

3.3. Verifying

After a Docker image is running, you can verify by using the classic `*nix` option `ps`. For example, issue the `$ docker ps -a` command.

```
[username ~]$ docker ps -a
CONTAINER ID      IMAGE                                COMMAND                  CREATED
12a4854ba738      nvcr.io/nvidia/tensorflow:21.02      "/usr/local/bin/nv..."   35 seconds ago
```

Without the `-a` option, only running instances are listed.

Important: It is best to include the `-a` option in case there are hung jobs running or other performance problems.

You can also stop a running container if you want. For example:

```
[username ~]$ docker ps -a
```

```
CONTAINER ID      IMAGE                                COMMAND                  PORTS     NAMES
12a4854ba738      nvcr.io/nvidia/tensorflow:21.02    "/usr/local/bin/nv..."   6006/tcp
 brave_neumann
[username ~]$
[username ~]$ docker stop 12a4854ba738
12a4854ba738
[username ~]$ docker ps -a
CONTAINER ID      IMAGE                                COMMAND                  CREATED
 NAMES
```

Notice that you need the *Container ID* of the image you want to stop. This can be found using the `$ docker ps -a` command.

Another useful command or Docker option is to remove the image from the server. Removing or deleting the image saves space on the server. For example, issue the following command:

```
$ docker rmi nvcr.io/nvidia.tensorflow:21.02
```

If you list the images, `$ docker images`, on the server, then you will see that the image is no longer there.

Chapter 4. NGC Images

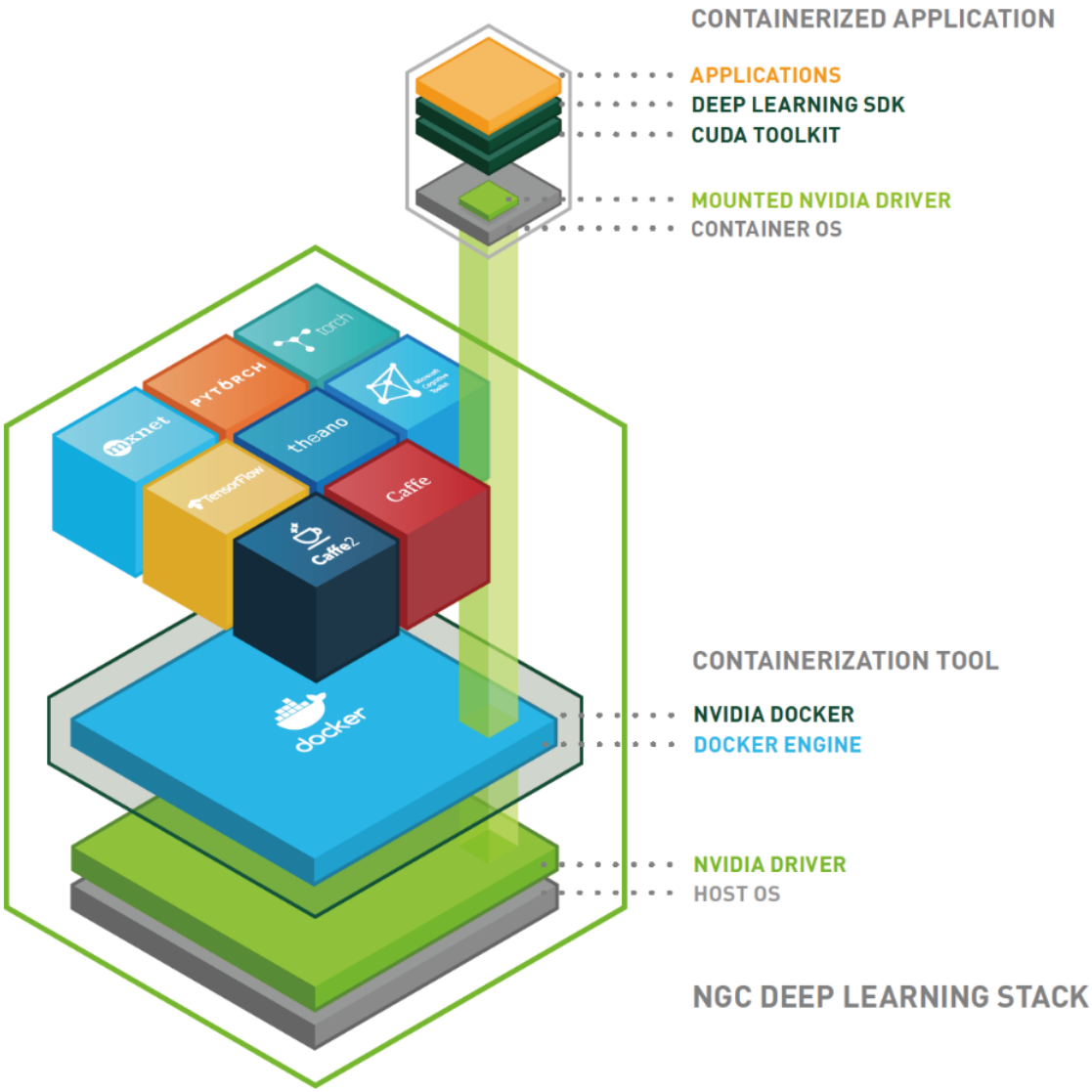
NGC containers are hosted in a repository called `nvcr.io`. As you read in the previous section, these containers can be “pulled” from the repository and used for GPU accelerated applications such as scientific workloads, visualization, and deep learning.

A Docker image is simply a file-system that a developer builds. Each layer depends on the layer below it in the stack.

From a Docker image, a container is created when the docker image is “run” or instantiated . 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.

Figure 1 depicts the stack for the DGX family of systems. Notice that the NVIDIA Container Toolkit sits above the host OS and the NVIDIA Drivers. The tools are used to create, manage, and use NVIDIA containers - these are the layers above the `nvidia-docker` 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. Docker containers encapsulate application dependencies to provide reproducible and reliable execution. The nvidia-docker utility mounts the user mode components of the NVIDIA driver and the GPUs into the Docker container at launch.



4.1. NGC Images Versions

Each release of a Docker 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 using tags defined by the year and month of the image release. For example, the 21.02 release of an image was released in February, 2021.

A complete 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.

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:21.01`

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.

Chapter 5. Running A Container

Before you begin

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.

5.1. Enabling GPU Support For 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 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.

Table 1. GPU Support For NGC Containers

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 version.

GPU Support Method	When Used	How To Determine
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

Note: 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 ...
```

5.2. Example: Running A Container

Procedure

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>
```

Note: 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.

Example: The following example runs the February 2021 release (21.02) of the NVIDIA PyTorch container in interactive mode. The container is automatically removed when the user exits the container.

```
$ docker run --gpus all --rm -ti nvcr.io/nvidia/pytorch:21.02-py3

==================
== NVIDIA PyTorch ==
==================
```

```
NVIDIA Release 21.02 (build 11032)

Container image Copyright (c) 2021, NVIDIA CORPORATION.  All rights reserved.
Copyright (c) 2014 - 2019, 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.

   Example: The following example runs the `pytorch time` command on one GPU to measure the execution time of the `deploy.prototxt` model.

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

3. **Optional:** Run the February 2021 release (21.02) of the same NVIDIA PyTorch container but in non-interactive mode.

   ```
   % docker run --gpus all -it --rm -v local_dir:container_dir nvcr.io/nvidia/
   pytorch:<xx.xx>-py3 <command>
   ```

5.3. 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>:<container
 version>
```

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.

5.4. 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 containers after exiting.

Note: 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>:<container version>
```

5.5. 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>:<container version>
```

5.6. Setting The Volumes Flag

There are no datasets 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>:<container version>
```

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>:<container
 version>
```

5.7. 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.

5.8. Setting The Shared Memory Flag

Certain applications, such as PyTorch™ , use shared memory buffers to communicate between processes. Shared memory can also be required by single process applications, such as NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet™ and TensorFlow™ , which use the NVIDIA® Collective Communications Library ™ (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.

5.9. 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`.

5.10. Container Lifetime

The state of an exited container is preserved indefinitely if you do not pass the `--rm` flag to the `docker run --gpus` 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 an 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 the 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 PyTorch container, issue this command:

  ```
  $ docker restart pytorch
  ```

- 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 .

Note: 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 6. NVIDIA Deep Learning Software Stack

The NVIDIA Deep Learning Software Developer Kit (SDK) contains everything that is on the NVIDIA registry area for DGX systems; including CUDA Toolkit, DIGITS and all of the deep learning frameworks.

The NVIDIA Deep Learning SDK accelerates widely-used deep learning frameworks such as NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet, PyTorch, and TensorFlow.

Note: Starting in the 18.09 container release, the Caffe2, Microsoft Cognitive Toolkit, Theano™ , and Torch™ frameworks are no longer provided within a container image.

The software stack provides containerized versions of these frameworks optimized for the system. These frameworks, including all necessary dependencies, are pre-built, tested, tuned, and ready to run. For users who need more flexibility to build custom deep learning solutions, each framework container image also includes the framework source code to enable custom modifications and enhancements, along with the complete software development stack.

The design of the platform software is centered around a minimal OS and driver installed on the server, and provisioning of all application and SDK software in the containers through the NGC container registry for DGX systems.

All NGC container images are based on the 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 applications.

6.1. 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.

6.2. CUDA Layer

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. For more information about CUDA, see the CUDA documentation.

6.2.1. 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.

6.2.2. CUDA Toolkit

The CUDA Toolkit provides a development environment for developing optimized GPU-accelerated applications. With the CUDA Toolkit, you can develop, optimize and deploy your applications to GPU-accelerated embedded systems, desktop workstations, enterprise data-centers and the cloud. The CUDA Toolkit includes libraries, tools for debugging and optimization, a compiler and a runtime library to deploy your application.

The following library provides GPU-accelerated primitives for deep neural networks:

CUDA® Basic Linear Algebra Subroutines library™ (cuBLAS) (cuBLAS)

cuBLAS is a GPU-accelerated version of the complete standard BLAS library that delivers significant speedup running on GPUs. The cuBLAS generalized matrix-matrix multiplication (GEMM) routine is a key computation used in deep neural networks, for example in computing fully connected layers. For more information about cuBLAS, see the cuBLAS documentation.

6.3. Deep Learning Libraries Layer

The following libraries are critical to deep learning on NVIDIA GPUs. These libraries are a part of the NVIDIA Deep Learning Software Development Kit (SDK).

6.3.1. NCCL

The NVIDIA® Collective Communications Library ™ (NCCL) (NCCL, pronounced “Nickel”) is a library of multi-GPU collective communication primitives that are topology-aware and can be easily integrated into applications.

Collective communication algorithms employ many processors working in concert to aggregate data. NCCL is not a full-blown parallel programming framework; rather, it is

a library focused on accelerating collective communication primitives. The following collective operations are currently supported:

- `AllReduce`
- `Broadcast`
- `Reduce`
- `AllGather`
- `ReduceScatter`

Tight synchronization between communicating processors is a key aspect of collective communication. CUDA based collectives would traditionally be realized through a combination of CUDA memory copy operations and CUDA kernels for local reductions. NCCL, on the other hand, implements each collective in a single kernel handling both communication and computation operations. This allows for fast synchronization and minimizes the resources needed to reach peak bandwidth.

NCCL conveniently removes the need for developers to optimize their applications for specific machines. NCCL provides fast collectives over multiple GPUs both within and across nodes. It supports a variety of interconnect technologies including PCIe, NVLink™ , InfiniBand Verbs, and IP sockets. NCCL also automatically patterns its communication strategy to match the system's underlying GPU interconnect topology.

Next to performance, ease of programming was the primary consideration in the design of NCCL. NCCL uses a simple C API, which can be easily accessed from a variety of programming languages. NCCL closely follows the popular collectives API defined by MPI (Message Passing Interface). Anyone familiar with MPI will thus find NCCL's API very natural to use. In a minor departure from MPI, NCCL collectives take a "stream" argument which provides direct integration with the CUDA programming model. Finally, NCCL is compatible with virtually any multi-GPU parallelization model, for example:

- single-threaded
- multi-threaded, for example, using one thread per GPU
- multi-process, for example, MPI combined with multi-threaded operation on GPUs

NCCL has found great application in deep learning frameworks, where the `AllReduce` collective is heavily used for neural network training. Efficient scaling of neural network training is possible with the multi-GPU and multi node communication provided by NCCL.

For more information about NCCL, see the NCCL documentation.

6.3.2. cuDNN Layer

The CUDA® Deep Neural Network library™ (cuDNN) (cuDNN) provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

Frameworks do not all progress at the same rate and the lack of backward compatibility within the cuDNN library forces it to be in its own container. This means that there will

be multiple CUDA and cuDNN containers available, but they will each have their own tag which the framework will need to specify in its Dockerfile.

For more information about cuDNN, see the cuDNN documentation.

6.4. Framework Containers

The framework layer includes all of the requirements for the specific deep learning framework. The primary goal of this layer is to provide a basic working framework. The frameworks can be further customized by a Platform Container layer specification.

Within the frameworks layer, you can choose to:

- Run a framework exactly as delivered by NVIDIA; in which case, the framework is built and ready to run inside that container image.
- Start with the framework as delivered by NVIDIA and modify it a bit; in which case, you can start from NVIDIA's container image, apply your modifications and recompile it inside the container.
- Start from scratch and build whatever application you want on top of the CUDA and cuDNN and NCCL layer that NVIDIA provides.

In the next section, the NVIDIA deep learning framework containers are presented.

For more information about frameworks, see the frameworks documentation.

Chapter 7. NVIDIA Deep Learning Framework Containers

A deep learning framework is part of a software stack that consists of several layers. Each layer depends on the layer below it in the stack. This software architecture has many advantages:

- Because each deep learning framework is in a separate container, each framework can use different versions of libraries such as the C standard library known as `libc`, cuDNN, and others, and not interfere with each other.
- A key reason for having layered containers is that one can target the experience for what the user requires.
- As deep learning frameworks are improved for performance or bug fixes, new versions of the containers are made available in the registry.
- The system is easy to maintain, and the OS image stays clean since applications are not installed directly on the OS.
- Security updates, driver updates and OS patches can be delivered seamlessly.

The following sections present the framework containers that are in `nvcr.io`.

7.1. Why Use a DL/ML Software Framework?

Frameworks have been created to make researching and applying deep learning more accessible and efficient. The key benefits of using frameworks include:

- Frameworks provide highly optimized GPU enabled code specific to the computations required for training Deep Neural Networks (DNN).
- NVIDIA frameworks are tuned and tested for the best possible GPU performance.
- Frameworks provide access to code through simple command line or scripting language interfaces such as Python.
- Many powerful DNNs can be trained and deployed using these frameworks without ever having to write any GPU or complex compiled code but while still benefiting from the training speed-up afforded by GPU acceleration.

7.2. Kaldi

The Kaldi Speech Recognition Toolkit project began in 2009 at Johns Hopkins University with the intent of developing techniques to reduce both the cost and time required to build speech recognition systems. While originally focused on ASR support for new languages and domains, the Kaldi project has steadily grown in size and capabilities, enabling hundreds of researchers to participate in advancing the field. Now the de-facto speech recognition toolkit in the community, Kaldi helps to enable speech services used by millions of people every day.

For information about the optimizations and changes that have been made to Kaldi, see the Deep Learning Frameworks Kaldi Release Notes.

7.3. NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet

NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet is a deep learning framework designed for both efficiency and flexibility, which allows you to mix the symbolic and imperative programming to maximize efficiency and productivity. MXNet is part of the Apache Incubator project. The MXNet library is portable and can scale to multiple GPUs and multiple machines. MXNet is supported by major Public Cloud providers including AWS and Azure Amazon; who have chosen MXNet as its deep learning framework of choice at AWS. It supports multiple languages (C++,Python,Julia,Matlab,JavaScript,Go,R,Scala,Perl,Wolfram Language).

At the core of NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet is a dynamic dependency scheduler that automatically parallelizes both symbolic and imperative operations on the fly. A graph optimization layer on top of the scheduler makes symbolic execution fast and memory efficient. NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet is portable and lightweight, and scales to multiple GPUs and multiple machines.

For information about the optimizations and changes that have been made to NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet, see the NVIDIA Optimized Deep Learning Framework, powered by Apache MXNet Release Notes.

7.4. TensorFlow

TensorFlow™ is an open-source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them.

This flexible architecture lets you deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

For visualizing TensorFlow results, this particular Docker image also contains TensorBoard. TensorBoard is a suite of visualization tools. For example, you can view the training histories as well as what the model looks like.

For information about the optimizations and changes that have been made to TensorFlow, see the Deep Learning Frameworks Release Notes.

7.4.1. Running The TensorFlow Container

An efficient way to run TensorFlow on the GPU system involves setting up a launcher script to run the code using a TensorFlow Docker container. For an example of how to run CIFAR-10 on multiple GPUs on system using `cifar10_multi_gpu_train.py`, see TensorFlow models.

If you prefer to use a script for running TensorFlow, see the `run_tf_cifar10.sh` script in the run_tf_cifar10.sh section. It is a bash script that you can run on a system. It assumes you have pulled the Docker container from the `nvcr.io` repository to the system. It also assumes you have the CIFAR-10 data stored in `/datasets/cifar` on the system and are mapping it to `/datasets/cifar` in the container. You can also pass arguments to the script such as the following:

```
$./run_tf_cifar10.sh --data_dir=/datasets/cifar --num_gpus=8
```

The details of the `run_tf_cifar10.sh` script parameterization is explained in the Keras section of this document (see Keras And Containerized Frameworks). You can modify the `/datasets/cifar` path in the script for the site specific location to CIFAR data. If the CIFAR-10 dataset for TensorFlow is not available, then run the example with writeable volume `-v /datasets/cifar:/datasets/cifar` (without `ro`) and the data will be downloaded on the first run.

If you want to parallelize the CIFAR-10 training, basic data-parallelization for TensorFlow via Keras can be done as well. Refer to the example `cifar10_cnn_mgpu.py` on GitHub.

A description of orchestrating a Python script with Docker containers is described in the run_tf_cifar10.sh script.

7.5. PyTorch

PyTorch is a GPU accelerated tensor computational framework with a Python front end. PyTorch is designed to be deeply integrated with Python. It is used naturally as you would use NumPy, SciPy and scikit-learn, or any other Python extension. You can even write the neural network layers in Python using libraries such as Cython and Numba. Acceleration libraries such as NVIDIA's cuDNN and NCCL, along with Intel's MKL are included to maximize performance.

PyTorch also includes standard defined neural network layers, deep learning optimizers, data loading utilities, and multi-GPU and multi-node support. Functions are executed immediately instead of enqueued in a static graph, improving ease of use and a sophisticated debugging experience.

For information about the optimizations and changes that have been made to PyTorch, see the Deep Learning Frameworks PyTorch Release Notes.

7.6. DIGITS

The Deep Learning GPU Training System™ (DIGITS) puts the power of deep learning into the hands of engineers and data scientists.

DIGITS is a popular training workflow manager provided by NVIDIA. Using DIGITS, one can manage image data sets and training through an easy to use web interface for the NVCaffe, Torch, and TensorFlow frameworks.

DIGITS is not a framework. DIGITS is a wrapper for NVCaffe, Torch and TensorFlow; which provides a graphical web interface to those frameworks rather than dealing with them directly on the command-line.

DIGITS can be used to rapidly train highly accurate DNNs for image classification, segmentation and object detection tasks. DIGITS simplifies common deep learning tasks such as managing data, designing and training neural networks on multi-GPU systems, monitoring performance in real time with advanced visualizations, and selecting the best performing model from the results browser for deployment. DIGITS is completely interactive so that data scientists can focus on designing and training networks rather than programming and debugging.

For information about the optimizations and changes that have been made to DIGITS, see the DIGITS Release Notes.

7.6.1. Setting Up DIGITS

The following directories, files and ports are useful in running the DIGITS container.

Table 2. Running DIGITS container details

Description	Value	Notes
DIGITS working directory	`$HOME/digits_workdir`	You must create this directory.
DIGITS job directory	`$HOME/digits_workdir/jobs`	You must create this directory.
DIGITS config file	`$HOME/digits_workdir/digits_config_env.sh`	Used to pass job directory and log file.

Description	Value	Notes
DIGITS port	5000	Choose a unique port if multi-user.

Important: It is recommended to specify a list of environment variables in a single file that can be passed to the `docker run --gpus` command via the `--env-file` option.

The digits_config_env.sh script declares the location of the DIGITS job directory and log file. This script is very popular when running DIGITS. Below is an example of defining these two variables in the simple bash script.

```
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
```

For more information about configuring DIGITS, see Configuration.md.

7.6.2. Running DIGITS

To run DIGITS, refer to the run_digits.sh script. However, if you want to run DIGITS from the command line, there is a simple command that has most of the needed details to effectively run DIGITS.

Note: You will have to create the `jobs` directory if it doesn't already exist.

```
$ mkdir -p $HOME/digits_workdir/jobs

$ NV_GPU=0,1 docker run --gpus all --rm -ti --name=${USER}_digits -p 5000:5000 \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  --env-file=${HOME}/digits_workdir/digits_config_env.sh \
  -v /datasets:/digits_data:ro \
  --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
  nvcr.io/nvidia/digits:17.05
```

This command has several options of which you might need, but you may not need all of them. The following table lists the parameters and their descriptions.

Table 3. `docker run --gpus` command options

Parameter	Description
NV_GPU	Optional environment variable specifying GPUs available to the container.
--name	Name to associate with the Docker container instance.
--rm	Tells Docker to remove the container instance when done.
-ti	Tells Docker to run in interactive mode and associate tty with the instance.
-d	Tells Docker to run in daemon mode; no tty, run in background (not shown in the command

Parameter	Description
	and not recommended for running with DIGITS).
`-p p1:p2`	Tells Docker to map host port `p1` to container port `p2` for external access. This is useful for pushing DIGITS output through a firewall.
`-u id:gid`	Tells Docker to run the container with user `id` and group `id` for file permissions.
`-v d1:d2`	Tells Docker to map host directory `d1` into the container at directory `d2`. **Important:** This is a very useful option because it allows you to store the data outside of the container.
`--env-file`	Tells Docker which environment variables to set for the container.
`--shm-size ...`	This line is a temporary workaround for a DIGITS multi-GPU error you might encounter.
`container`	Tells Docker which container instance to run (for example, `nvcr.io/nvidia/digits:17.05`).
`command`	Optional command to run after the container is started. This option is not used in the example.

After DIGITS starts running, open a browser using the IP address and port of the system. For example, the URL would be `http://dgxip:5000/`. If the port is blocked and an SSH tunnel has been set up, then you can use the URL `http://localhost:5000/`.

In this example, the datasets are mounted to `/digits_data` (inside the container) via the option `-v /datasets:/digits_data:ro`. Outside the container, the datasets reside in `/datasets` (this can be any path on the system). Inside the container the data is mapped to `/digits_data`. It is also mounted read-only (`ro`) with the option `:ro`.

Important: For both paths, it is highly recommended to use the fully qualified path name for outside the container and inside the container.

If you are looking for datasets for learning how to use the system and the containers, there are some standard datasets that can be downloaded via DIGITS.

Included in the DIGITS container is a Python script that can be used to download specific sample datasets. The tool is called `digits.download_data`. It can be used to download the MNIST data set, the CIFAR-10 dataset, and the CIFAR-100 dataset. You can also use this script in the command line to run DIGITS so that it pulls down the sample dataset. Below is an example for the MNIST dataset.

```
docker run --gpus all --rm -ti \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  --env-file=${HOME}/digits_workdir/digits_config_env.sh \
  -v /datasets:/digits_data \
  --entrypoint=bash \
  nvcr.io/nvidia/digits:17.05 \
  -c 'python -m digits.download_data mnist /digits_data/digits_mnist'
```

In the download example above, the entry point to the container was overridden to run a bash command to download the dataset (the `-c` option). You should adjust the datasets paths as needed.

An example of running DIGITS on MNIST data can be found here.

More DIGITS examples can be found here.

Chapter 8. Frameworks General Best Practices

As part of DGX systems, NVIDIA makes available tuned, optimized, tested, and ready to run Docker containers for the major deep learning frameworks. These containers are made available via the NGC container registry, `nvcr.io`, so that you can use them directly or use them as a basis for creating your own containers.

This section presents tips for efficiently using these frameworks. For best practices regarding how to use Docker, see Docker And Container Best Practices. To get started with NVIDIA containers, see Preparing To Use NVIDIA Containers.

8.1. Extending Containers

There are a few general best practices around the containers (the frameworks) in `nvcr.io`. As mentioned earlier, it's possible to use one of the containers and build upon it (extend it). By doing this, you are in a sense fixing the new container to a specific framework and container version. This approach works well if you are creating a derivative of a framework or adding some capability that doesn't exist in the framework or container.

However, if you extend a framework understand that in a few months time, the framework will have likely changed. This is due to the speed of development of deep learning and deep learning frameworks. By extending a specific framework, you have locked the extensions into that particular version of the framework. As the framework evolves, you will have to add your extensions to these new versions, increasing your workload. If possible, it's highly recommended to not tie the extensions to a specific container but keep them outside. If the extensions are invasive, then it is recommended to discuss the patches with the framework team for inclusion.

8.2. Datasets And Containers

You might be tempted to extend a container by putting a dataset into it. But once again, you are now fixing that container to a specific version. If you go to a new version of a framework or a new framework you will have to copy the data into it. This makes keeping up with the fast paced development of frameworks very difficult.

A best practice is to not put datasets in a container. If possible also avoid storing business logic code in a container. The reason is because by storing datasets or business logic code within a container, it becomes difficult to generalize the usage of the container.

Instead, one can mount file systems into a container that contain only the desired data sets and directories with business logic code to run. Decoupling the container from specific datasets and business logic enables you to easily change containers, such as framework or version of a container, without having to rebuild the container to hold the data or code.

The subsequent sections briefly present some best practices around the major frameworks that are in containers on the container registry (`nvcr.io`). There is also a section that discusses how to use Keras, a very popular high-level abstraction of deep learning frameworks, with some of the containers.

8.3. Keras And Containerized Frameworks

Keras is a popular Python frontend for TensorFlow, Theano, and Microsoft Cognitive Toolkit v.2.x release. Keras implements a high-level neural network API to the frameworks listed. Keras is not included in the containers in `nvcr.io` because it is evolving so quickly. You can add it to any of the containers if you like, but there are ways to start one of the `nvcr.io` containers and install Keras during the launch process. This section also provides some scripts for using Keras in a virtual Python environment.

Before jumping into Keras and best practices around how to use it, a good background for Keras is to familiarize yourself with virtualenv and virtualenvwrapper.

When you run Keras, you have to specify the desired framework backend. This can be done using either the `$HOME/.keras/keras.json` file or by an environment variable `KERAS_BACKEND=<backend>` where the backend choices are: `theano`, `tensorflow`, or `cntk`. The ability to choose a framework with minimal changes to the Python code makes Keras very popular.

There are several ways to configure Keras to work with containerized frameworks.

Important: The most reliable approach is to create a container with Keras or install Keras within a container.

Setting up a container with Keras might be preferable for deployed containerized services.

Important: Another approach that works well in development environments is to setup a virtual Python environment with Keras.

This virtual environment can then be mapped into the container and the Keras code can run against the desired framework backend.

The advantage of decoupling Python environments from the containerized frameworks is that given `M` containers and `N` environments instead of having to create `M * N` containers, one can just create `M + N` configurations. The configuration then is the launcher or orchestration script that starts the desired container and activates the Keras Python environment within that container. The disadvantage with such an approach is that one cannot guarantee the compatibility of the virtual Python environment and the framework backend without testing. If the environment is incompatible then one would need to re-create the virtual Python environment from within the container to make it compatible.

8.3.1. Adding Keras To Containers

If you choose, you can add Keras to an existing container. Like the frameworks themselves, Keras changes fairly rapidly so you will have to watch for changes in Keras.

There are two good choices for installing Keras into an existing container. Before proceeding with either approach, ensure you are familiar with the Preparing to Use Docker Containers guide to understand how to build on existing containers.

The first approach is to use the OS version of Python to install Keras using the Python tool `pip`.

```
# sudo pip install keras
```

Ensure you check the version of Keras that has been installed. This may be an older version to better match the system OS version but it may not be the version you want or need. If that is the case, the next paragraph describes how to install Keras from source code.

The second approach is to build Keras from source. It is recommended that you download one of the releases rather than download from the main branch. A simple step-by-step process is to:

1. Download a release in `.tar.gz` format (you can always use `.zip` if you want).
2. Start up a container with TensorFlow.
3. Mount your home directory as a volume in the container (see Using And Mounting File Systems).
4. Navigate into the container and open a shell prompt.
5. Uncompress and untar the Keras release (or unzip the `.zip` file).
6. Issue `cd` into the directory.

```
# cd keras
# sudo python setup.py install
```

If you want to use Keras as part of a virtual Python environment, the next section will explain how you can achieve that.

8.3.2. Creating Keras Virtual Python Environment

Before jumping into Keras in a virtual Python environment, it's always a good idea to review the installation dependencies of Keras. The dependencies are common for data science Python environments, NumPy, SciPy, YAML, and h5py. It can also use cuDNN, but this is already included in the framework containers.

You will be presented with several scripts for running Keras in a virtual Python environment. These scripts are included in the document and provides a better user experience than having to do things by hand.

The venvfns.sh script needs to be put in a directory on the system that is accessible for all users, for example, it could be placed in `/usr/share/virtualenvwrapper/`. An administrator needs to put this script in the desired location since it has to be in a directory that every user can access.

The setup_keras.sh script creates a `py-keras` virtual Python environment in `~/.virtualenvs` directory (this is in the user's home directory). Each user can run the script as:

```
$ ./setup_keras.sh
```

In this script, you launch the `nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04` container as the local user with your home directory mounted into the container. The salient parts of the script are below:

```
dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04
```

Important: When creating the Keras files, ensure you have the correct privileges set when using the `-u` or `--user` options. The `-d` and `-t` options daemonize the container process. This way the container runs in the background as a daemon service and one can execute code against it.

You can use `docker exec` to execute a snippet of code, a script, or attach interactively to the container. Below is the portion of the script that sets up a Keras virtual Python environment.

```
docker exec -it $dname \
  bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
  mkvirtualenv py-keras
  pip install --upgrade pip
  pip install keras --no-deps
  pip install PyYaml
  # pip install -r /pathto/requirements.txt
  pip install numpy
  pip install scipy
  pip install ipython'
```

If the list of Python packages is extensive, you can write a `requirements.txt` file listing those packages and install via:

```
pip install -r /pathto/requirements.txt --no-deps
```

Note: This particular line is in the previous command, however, it has been commented out because it was not needed.

The `--no-deps` option specifies that dependencies of packages should not be installed. It is used here because by default installing Keras will also install TensorFlow.

Important: On a system where you don't want to install non-optimized frameworks such as TensorFlow, the `--no-deps` option prevents this from happening.

Notice the line in the script that begins with `bash -c ....` This points to the script previously mentioned (`venvfns.sh`) that needs to be put in a common location on the system. If some time later, more packages are needed, one can relaunch the container and add those new packages as above or interactively. The code snippet below illustrates how to do so interactively.

```
dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04

sleep 2  # wait for above container to come up

docker exec -it $dname bash
```

You can now log into the interactive session where you activated the virtual Python environment and install what is needed. The example below installs h5py which is used by Keras for saving models in HDF5 format.

```
source ~/.virtualenvs/py-keras/bin/activate
pip install h5py
deactivate
exit
```

If the installation fails because some underlying library is missing, one can attach to the container as root and install the missing library.

The next example illustrates installing the `python-dev` package which will install `Python.h` if it is missing.

```
$ docker exec -it -u root $dname \
  bash -c 'apt-get update &&  apt-get install -y python-dev # anything else...'
```

The container can be stopped or removed when you are done using the following command.

```
$ docker stop $dname && docker rm $dname
```

8.3.3. Using Keras Virtual Python Environment With Containerized Frameworks

The following examples assume that a `py-keras venv` (Python virtual environment) has been created per the instructions in the previous section. All of the scripts for this section can be found in the Scripts section.

The run_kerastf_mnist.sh script demonstrates how the Keras `venv` is enabled and is then used to run the Keras MNIST code `mnist_cnn.py` with the default backend TensorFlow. Standard Keras examples can be found here.

Compare the run_kerastf_mnist.sh script to the run_kerasth_mnist.sh that uses Theano. There are primarily two differences:

1. The backend container `nvcr.io/nvidia/theano:17.05` is used instead of `nvcr.io/nvidia/tensorflow:17.05`.
2. In the code launching section of the script, specify `KERAS_BACKEND=theano`. You can run these scripts as:

   ```
   $./run_kerasth_mnist.sh  # Ctrl^C to stop running
   $./run_kerastf_mnist.sh
   ```

The run_kerastf_cifar10.sh script has been modified to accept parameters and demonstrates how one would specify an external data directory for the CIFAR-10 data. The cifar10_cnn_filesystem.py script has been modified from the original `cifar10_cnn.py`. The command line example to run this code on a system is the following:

```
$./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
```

The above assumes the storage is mounted on a system at `/datasets/cifar`.

Important: The key takeaway is that running some code within a container involves setting up a launcher script.

These scripts can be generalized and parameterized for convenience and it is up to the end user or developer to write these scripts for their custom application or their custom workflow.

For example:

1. The parameters in the example script were joined to a temporary variable via the following:
   ```
   function join { local IFS="$1"; shift; echo "$*"; }
   script_args=$(join : "$@")
   ```
2. The parameters were passed to the container via the option:
   ```
   -e script_args="$script_args"
   ```
3. Within the container, these parameters are split and passed through to the computation code by the line:
   ```
   python $cifarcode ${script_args//:/ }
   ```
4. The external system NFS/storage was passed as read-only to the container via the following option to the launcher script:
   ```
   -v /datasets/cifar:/datasets/cifar:ro
   ```
   and by
   ```
   --datadir=/datasets/cifar
   ```

The run_kerastf_cifar10.sh script can be improved by parsing parameters to generalize the launcher logic and avoid duplication. There are several ways to parse parameters in bash via `getopts` or a custom parser. One can write a non-bash launcher as well as using Python, Perl, or something else.

The run_keras_script script implements a high-level parameterized bash launcher. The following examples illustrate how to use it to run the previous MNIST and CIFAR examples above.

```
# running Tensorflow MNIST
./run_keras_script.sh \
  --container=nvcr.io/nvidia/tensorflow:17.05 \
  --script=examples/keras/mnist_cnn.py

# running Theano MNIST
./run_keras_script.sh \
  --container=nvcr.io/nvidia/theano:17.05 --backend=theano \
  --script=examples/keras/mnist_cnn.py

# running Tensorflow Cifar10
./run_keras_script.sh \
  --container=nvcr.io/nvidia/tensorflow:17.05 --backend=tensorflow \
  --datamnt=/datasets/cifar \
  --script=examples/keras/cifar10_cnn_filesystem.py \
```

```
 --epochs=3 --datadir=/datasets/cifar

# running Theano Cifar10
./run_keras_script.sh \
  --container=nvcr.io/nvidia/theano:17.05 --backend=theano \
  --datamnt=/datasets/cifar \
  --script=examples/keras/cifar10_cnn_filesystem.py \
 --epochs=3 --datadir=/datasets/cifar
```

Important: If the code is producing output that needs to be written to a filesystem and persisted after the container stops, that logic needs to be added.

The examples above show containers where their home directory is mounted and is "writeable". This ensures that the code can write the results somewhere within the user's home path. The filesystem paths need to be mounted into the container and specified or passed to the computational code.

These examples serve to illustrate how one goes about orchestrating computational code via Keras or even non-Keras.

Important: In practice, it is often convenient to launch containers interactively, attach to them interactively, and run code interactively.

During these interactive sessions, it is easier to (automate via helper scripts) debug and develop code. An interactive session might look like the following sequence of commands typed manually into the terminal:

```
# in bash terminal
dname=mykerastf

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
  nvcr.io/nvidia/tensorflow:17.05

docker exec -it $dname bash
# now interactively in the container.
source ~/.virtualenvs/py-keras/bin/activate
source ~/venvfns.sh
enablevenvglobalsitepackages
./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# change some parameters or code in cifar10_cnn_filesystem.py and run again
./run_kerastf_cifar10.sh --aug --epochs=2 --datadir=/datasets/cifar
disablevenvglobalsitepackages
exit # exit interactive session in container

docker stop $dname && docker rm $dname # stop and remove container
```

8.3.4. Working With Containerized VNC Desktop Environment

The need for a containerized desktop varies depending on the data center setup. If the systems are set up behind a login node or a head node for an on-premise system, typically data centers will provide a VNC login node or run X Windows on the login node to facilitate running visual tools such as text editors or an IDE (integrated development environment).

For a cloud based system (NGC), there may already be firewalls and security rules available. In this case, you may want to ensure that the proper ports are open for VNC or something similar.

If the system serves as the primary resource for both development and computing, then it is possible to setup a desktop-like environment on it via containerized desktop. The instructions and `Dockerfile` for this can be found here.

You can download the latest release of the container to the system. The next step is to modify the `Dockerfile` by changing the `FROM` field to be:

```
FROM nvcr.io/nvidia/cuda:11.0-cudnn6-devel-ubuntu20.04
```

This is not an officially supported container by the NVIDIA DGX product team, in other words, it is not available on `nvcr.io` and was provided as an example of how to setup a desktop-like environment on a system for convenient development with `eclipse` or `sublime-text` (suggestion, try visual studio code which is very like sublime text but free) or any other GUI driven tool.

The `build_run_dgxdesk.sh` example script is available on the GitHub site to build and run a containerized desktop as shown in the Scripts section. Other systems such as the DGX Station and NGC would follow a similar process.

To connect to the system, you can download a VNC client for your system from RealVnc, or use a web-browser.

```
=> connect via VNC viewer hostip:5901, default password: vncpassword
=> connect via noVNC HTML5 client: http://hostip:6901/?password=vncpassword
```

Chapter 9. HPC And HPC Visualization Containers

HPC Visualization Containers

In addition to accessing the NVIDIA optimized frameworks and HPC containers, the NGC container registry also hosts scientific visualization containers for HPC. These containers rely on the popular scientific visualization tool called ParaView.

Visualization in an HPC environment typically requires remote visualization, that is, data resides and is processed on a remote HPC system or in the cloud, and the user graphically interacts with this application from their workstation. As some visualization containers require specialized client applications, the HPC visualization containers consist of two components:

Server container

The server container needs access to the files on your server system. Details on how to grant this access are provided below. The server container can run both in serial mode or in parallel. For this alpha release, we are focusing on the serial node configuration. If you are interested in parallel configuration, contact hpcviscontainer@nvidia.com.

Client container

To ensure matching versions of the client application and the server container, NVIDIA provides the client application in a container. Similarly, to the server container, the client container needs access to some of the ports to establish connection with the server container.

In addition, the client container needs access to the users' X server for displaying the graphical user interface.

NVIDIA recommends to map a host file system into the client container in order to enable saving of the visualization products or other data. In addition, the connection between the client and server container needs to be opened.

For a list of available HPC visualization containers and steps on how to use them, see the NGC Container User Guide.

Chapter 10. Customizing And Extending Containers And Frameworks

NVIDIA Docker 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 DGX systems, you can push or save your modified/extended containers to the NGC container registry, `nvcr.io`. They can also be shared with other users of the DGX system but this requires some administrator help.

It is important to note that all 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.

10.1. Customizing A Container

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.

10.1.1. 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.

10.1.2. Example 1: Building A Container From Scratch

About this task

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 even if you are just using a base OS.

For best practices on writing Dockerfiles, see Best practices for writing Dockerfiles.

As an example, let's create a container from a Dockerfile that uses Ubuntu 20.04 as a base OS. Let's also update the OS when we create our container.

Procedure

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:20.04
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.

   In this example, we are also pulling the container from the Docker repository and not the NGC 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> .
```

Note: 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's name the container `test1` and tag it with `latest`. Also, for illustrative purposes, let's assume our private DGX system repository is called `nvidian_sas` (the exact name depends upon how you registered the DGX. This is typically the company name in some fashion.) 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 8.012 kB
Step 1/3 : FROM ubuntu:20.04
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 1f391b9285d8
 ---> 934785072daf
Removing intermediate container 1f391b9285d8
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.

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

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

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

Note: You need to have a DGX system to do this.

8. Store the container in your private Docker repository by pushing it.
   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, for example, to 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.

10.1.3. Example 2: Customizing A Container Using Dockerfile

About this task

This example uses a Dockerfile to customize the PyTorch container in `nvcr.io`. Before customizing the container, you should ensure the PyTorch 21.02 container has been loaded into the registry using the `docker pull` command before proceeding.

```
$ docker pull nvcr.io/nvidia/pytorch:21.02-py3
```

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.

Procedure

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/pytorch:21.02
# APPLY CUSTOMER PATCHES TO PYTORCH
# Bring in changes from outside container to /tmp
# (assumes my-pytorch-modifications.patch is in same directory as
Dockerfile)
#COPY my-pytorch-modifications.patch /tmp

# Change working directory to PyTorch source path
WORKDIR /opt/pytorch
```

```
# Apply modifications
#RUN patch -p1 < /tmp/my-pytorch-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/pytorch* and the tag is *21.02.1PlusChanges* .. For this case, the command would be the following:

```
$ docker build -t corp/pytorch:21.02.1PlusChanges .
```

5. Run the Docker image.

```
docker run --gpus all -ti --rm corp/pytorch:21.02.1PlusChanges .
```

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

About this task

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.

Note:

- 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.

Procedure

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:
   docker run --gpus all --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 ...

root@1fe228556a97:/workspace#
```

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.

Note: 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
```

Note: 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 a snippet of 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.

Note: 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:
   docker run --gpus all --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
```

Results

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

10.1.5. Example 4: Developing A Container Using Docker

About this task

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 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 through 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.

Note: 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.

Procedure

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
```

10.1.5.1. Example 4.1: Package The Source Into The Container

About this task

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.

10.2. Customizing aFramework

Each Docker image contains the code required to build the framework so that you can make changes to the framework itself. The location of the framework source in each image is in the `/workspace` directory.

For specific directory locations, see the Deep Learning Framework Release Notes for your specific framework.

10.2.1. Benefits and Limitations to Customizing a Framework

Customizing a framework is useful if you have patches or modifications you want to make to the framework outside of the NVIDIA repository or if you have a special patch that you want to add to the framework.

10.2.2. Example 1: Customizing A Framework Using The Command Line

About this task

This Dockerfile example illustrates a method to apply patches to the source code in the NVCaffe container image and to rebuild NVCaffe. The `RUN` command included below will rebuild NVCaffe in the same way as it was built in the original image.

By applying customizations through a Dockerfile and `docker build` in this manner rather than modifying the container interactively, it will be straightforward to apply the same changes to later versions of the NVCaffe container image.

For more information, see Dockerfile reference.

Procedure

1. Create a working directory for the Dockerfile.

```
$ mkdir docker
$ cd docker
```

2. Open a text editor and create a file called `Dockerfile` and add the following lines:

```
FROM  nvcr.io/nvidia/caffe:17.04
RUN apt-get update && apt-get install bc
```

3. Bring in changes from outside the container to `/tmp`.

Note:

This assumes `my-caffe-modifications.patch` is in same directory as Dockerfile.

```
COPY my-caffe-modifications.patch /tmp
```

4. Change your working directory to the NVCaffe source path.
```
WORKDIR /opt/caffe
```
5. Apply your modifications.
```
RUN patch -p1 < /tmp/my-caffe-modifications.patch
```
6. Rebuild NVCaffe.
```
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
```
7. Reset the default working directory.
```
WORKDIR /workspace
```

10.2.3. Example 2: Customizing A Framework And Rebuilding The Container

About this task

This example illustrates how you can customize a framework and rebuild the container. For this example, we will use the NVCaffe 17.03 framework.

Currently, the NVCaffe framework returns the following output message to `stdout` when a network layer is created:

```
"Creating Layer"
```

For example, you can see this output by running the following command from a bash shell in a NVCaffe 17.03 container.

```
# which caffe
/usr/local/bin/caffe
# caffe time --model /workspace/models/bvlc_alexnet/deploy.prototxt
--gpu=0
…
I0523 17:57:25.603410 41 net.cpp:161] Created Layer data (0)
I0523 17:57:25.603426 41 net.cpp:501] data -> data
I0523 17:57:25.604748 41 net.cpp:216] Setting up data
…
```

The following steps show you how to change the message "`Created Layer`" in NVCaffe to "`Just Created Layer`". This example illustrates how you might modify an existing framework.

Before you begin

Ensure you run the framework container in interactive mode.

Procedure

1. Locate the NVCaffe 17.03 container from the `nvcr.io` repository.
```
$ docker pull nvcr.io/nvidia/caffe:17.03
```
2. Run the container on the DGX system.

```
docker run --gpus all --rm -ti nvcr.io/nvidia/caffe:17.03
```

Note: This will make you the root user in the container. Notice the change in the prompt.

3. Edit a file in the NVCaffe source file, `/opt/caffe/src/caffe/net.cpp`. The line you want to change is around line `162`.

```
# vi /opt/caffe/src/caffe/net.cpp
:162 s/Created Layer/Just Created Layer
```

Note: This uses vi. Change `"Created Layer"` to `"Just Created Layer"`.

4. Rebuild NVCaffe.

```
# cd /opt/caffe
# 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"$(proc)" install
# make install
# ldconfig
```

5. Before running the updated NVCaffe framework, ensure the updated NVCaffe binary is in the correct location, for example, `/usr/local/`.

```
# which caffe
/usr/local/bin/caffe
```

6. Run NVCaffe and look for a change in the output to `stdout`:

```
# caffe time --model /workspace/models/bvlc_alexnet/deploy.prototxt
--gpu=0
/usr/local/bin/caffe
…
I0523 18:29:06.942697  7795 net.cpp:161] Just Created Layer data (0)
I0523 18:29:06.942711  7795 net.cpp:501] data -> data
I0523 18:29:06.944180  7795 net.cpp:216] Setting up data
...
```

7. Save your container to your private DGX repository on `nvcr.io` or your private Docker repository (see Example 2: Customizing A Container Using Dockerfile for an example).

10.3. Optimizing Docker Containers For Size

The Docker container format using layers was specifically designed to limit the amount of data that would need to be transferred when a container image is instantiated. When a Docker container image is instantiated or “pulled” from a repository, Docker may need to copy the layers from the repository to the local host. It checks what layers it already has on the host using the hash for each layer. If it already has it on the local host, it won’t ”re-download” it saving time, and to a smaller degree, network usage.

This is particularly useful for NVIDIA’s NGC because all the containers are built with the same base OS and libraries. If you run one container image from NGC, then run another, it is likely that many of the layers from the first container are used in the second container,

reducing the time to pull down the second container image so the container can be started quickly.

You can put almost anything you want into a container allowing users or container developers to create very large (GB+) containers. Even though it is not recommended to put data in your Docker container image, users and developers do this (there are some good reasons). This can further inflate the size of the container image. This increases the amount of time to download a container image or it's various layers. Users and developers are now asking for ways to reduce the size of the container image or the individual layers.

The following subsections present some options that you can use if the container image or the layer sizes are too large or you want them smaller. There is no single option that works best, so be sure to try them on your container images.

10.3.1. One Line Per RUN Command

In a Dockerfile, using one line for each RUN command is very convenient. The code is easy to read since you can see each command. However, Docker will create a layer for each command. Each layer keeps some information (metadata) about its origins, when the layer was created, what is contained in the layer, and a hash for each layer. If you have a large number of commands, you are going to have a large amount of metadata.

A simple way to reduce the size of the container image is to put all of the RUN commands that you can into a single RUN statement. This may result in a very large RUN command, however, it greatly reduces the amount of metadata. It is recommended that you group as many RUN commands together as possible. Depending upon your Dockerfile, you may not be able to put all RUN commands into a single RUN statement. Do your best to reduce the number of RUN commands but make it logical.

Below is a simple Dockerfile example used to build a container image.

```
$ cat Dockerfile
FROM ubuntu:20.04

RUN date > /build-info.txt
RUN uname -r >> /build-info.txt

Notice there are two RUN commands in this simple Dockerfile. The container image can be built
 using the following command and associated output.
$ docker build -t first-image -f Dockerfile .
…
Step 2/3 : RUN date > /build-info.txt
 ---> Using cache
 ---> af12c4b34f91
Step 3/3 : RUN uname -r >> /build-info.txt
 ---> Running in 0f883f37e3c8
…
```

Notice that the RUN commands each created a layer in the container image.

Let's examine the container image for details on the layers.

```
$ docker run --rm -it first-image cat /build-info.txt
Mon Jan 18 10:14:02 UTC 2021
5.5.115-1.el7.elrepo.x86_64

$ docker history first-image
IMAGE               CREATED             CREATED BY                                      SIZE

d2c03aa61290        11 seconds ago      /bin/sh -c uname -r >> /build-info.txt          57B
```

```
af12c4b34f91        16 minutes ago      /bin/sh -c date > /build-info.txt                29B

5e8b97a2a082        6 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]             0B

<missing>           6 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B

<missing>           6 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$…
 2.76kB
<missing>           6 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B

<missing>           6 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B

<missing>           6 weeks ago         /bin/sh -c #(nop) ADD file:d37ff24540ea7700d…
 114MB
```

The output of this command gives you information about each of the layers. Notice that there is a layer for each `RUN` command.

Now, let's take the Dockerfile and combine the two `RUN` commands.

```
$ cat Dockerfile
FROM ubuntu:20.04

RUN date > /build-info.txt && uname -r >> /build-info.txt
$ docker build -t one-layer -f Dockerfile .

$ docker history one-layer
IMAGE               CREATED             CREATED BY                                      SIZE

3b1ef5bc19b2        6 seconds ago       /bin/sh -c date > /build-info.txt && uname -…   57B

5e8b97a2a082        6 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B

<missing>           6 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B

<missing>           6 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$…
 2.76kB
<missing>           6 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B

<missing>           6 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B

<missing>           6 weeks ago         /bin/sh -c #(nop) ADD file:d37ff24540ea7700d…   114MB
```

Notice that there is now only one layer that has both `RUN` commands included.

Another good reason to combine `RUN` commands is that if you have multiple layers, it's easy to modify one layer in the container image without having to modify the entire container image.

10.3.2. Export, Import, And Flatten

If space is at a premium, there is a way to take the existing container image, and get rid of all the history. It can only be done using a running container. Once the container is running, run the following two commands:

```
# export the container to a tarball
docker export <CONTAINER ID> > /home/export.tar

# import it back
cat /home/export.tar | docker import - some-name:<tag>
```

This will get rid of the history of each layer but it will preserve the layers (if that is important).

Another option is to "flatten" your image to a single layer. This gets rid of all the redundancies in the layers and creates a single container. Like the previous technique,

this one requires a running container as well. With the container running, issue the following command:

```
docker export <CONTAINER ID> | docker import - some-image-name:<tag>
```

This pipeline exports the container through the `import` command creating a new container that is only one layer. For more information, see this blog post.

10.3.3. `docker-squash`

A few years ago before Docker, adding the ability to "squash" images via a tool called `docker-squash` was created. It hasn't been updated for a couple of years, however, it is still a popular tool for reducing the size of Docker container images. The tool takes a Docker container image and "squashes" it to a single layer, reducing commonalities between layers and history of the layers producing the smallest possible container image.

The tool retains Docker commands such as `PORT`, `ENV`, etc. the squashed images work exactly the same as before they were squashed. Moreover, the files that are deleted during the squashing process are actually removed from the image.

A simple example for running `docker-squash` is below.

```
docker save <ID> | docker-squash -t <TAG> [-from <ID>] | docker load
```

This pipeline takes the current image, saves it, squashes it with a new tag, and reloads the container. The resulting image has all the layers beneath the initial `FROM` layer squashed into a single layer. The default options in `docker-squash` retains the base image layer so that it does not need to be repeatedly transferred when pushing and pulling updates to the image.

The tool is really designed for containers that are finalized and not likely to be updated. Consequently, there is little need for details about the layers and history. It can then be squashed and put into production. Having the smallest size image will allow users to quickly download the image and get it running because it's almost as small as possible.

10.3.4. Squash While Building

Not long after Docker came out, people started creating giant images that took a long time to transfer. At that point, users and developers started working on ideas to reduce the container size. Not too long ago, some patches were proposed for Docker to allow it to squash images as they were being built. The `squash` option was added in Docker 1.13 (API 1.25), when Docker still followed a different versioning scheme. As of Docker 17.06-ce the option is still classified as experimental. You can tell Docker to allow the use of experimental options if you want (refer to Docker documentation). However, NVIDIA does not support this option.

The `--squash` option is used when the container is built. An example of the command is the following:

```
docker build --squash -t chamilad/testdocker:0.1 .
```

This command uses "Dockerfile" as the dockerfile for building the container.

The `--squash` option creates an image that has two layers. The first layer results from the `FROM` that usually starts off a Dockerfile. The subsequent layers are all "squashed"

together into a single layer. This gets rid of the history in all the layers but the first one. It also eliminates redundant files.

Since it is still an experimental feature, the amount you can squeeze the image varies. There have been reports of a 50% reduction in image size.

10.3.5. Additional Options

There are some other options that be used to reduce the size of images, but they are not particularly Docker based (although there are a couple). The rest are classic Linux commands.

There is a Docker build option that deals with building applications in Docker containers. If you want to build an application when the container is created, you may not want to leave the building tools in the image because of its size. This is true when the container is supposed to be executed and not modified when it is run. Recall that Docker containers are built in layers. We can use that fact when building containers to copy binaries from one layer to another.

For example, the Docker file below:

```
$ cat Dockerfile
FROM ubuntu:20.04

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
        build-essential \
        gcc && \
    rm -rf /var/lib/apt/lists/*

COPY hello.c /tmp/hello.c
RUN gcc -o /tmp/hello /tmp/hello.c
```

Builds a container, installs `gcc`, and builds a simple “hello world” application. Checking the history of the container will give us the size of the layers:

```
$ docker history hello
IMAGE               CREATED             CREATED BY                                      SIZE

49fef0e11806        8 minutes ago       /bin/sh -c gcc -o /tmp/hello /tmp/hello.c       8.6kB

44a449445055        8 minutes ago       /bin/sh -c #(nop) COPY file:8f0c1776b2571c38…   63B

c2e5b659a549        8 minutes ago       /bin/sh -c apt-get update -y &&     apt-get …   181MB

5e8b97a2a082        6 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B

<missing>           6 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B

<missing>           6 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$…
 2.76kB
<missing>           6 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B

<missing>           6 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B

<missing>           6 weeks ago         /bin/sh -c #(nop) ADD file:d37ff24540ea7700d…   114MB
```

Notice that the layer with the build tools is 181MB in size, yet the application layer is only 8.6kB in size. If the build tools aren’t needed in the final container, then we can get rid of it from the image. However, if you simply do a `apt-get remove …` command, the build tools are not actually erased.

A solution is to copy the binary from the previous layer to a new layer as in this Dockerfile:

```
$ cat Dockerfile
FROM ubuntu:16.04 AS build

RUN apt-get update -y && \
    apt-get install -y --no-install-recommends \
        build-essential \
        gcc && \
    rm -rf /var/lib/apt/lists/*

COPY hello.c /tmp/hello.c

RUN gcc -o /tmp/hello /tmp/hello.c

FROM ubuntu:16.04

COPY --from=build /tmp/hello /tmp/hello
```

This can be termed a “multi-stage” build. In this Dockerfile, the first stage starts with the OS and names it “build”. Then the build tools are installed, the source is copied into the container, and the binary is built.

The next layer starts with a fresh OS `FROM` command (referred to as a “first stage”). Docker will only save the layers starting with this one and any subsequent layers (in other words, the first layers that installed the build tools won’t be saved) or the “second stage”. The second stage can copy the binary from the first stage. No build tools are included in this stage. Building the container image is the same as before.

If we compare the size of the container with the first Dockerfile to the size using the second Dockerfile, we can see the following:

```
$ docker images hello
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello               latest              49fef0e11806        21 minutes ago      295MB
$ docker images hello-rt
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
hello-rt            latest              f0cef59a05dd        2 minutes ago       114MB
```

The first output is the original Dockerfile. The second output is for the multistage Dockerfile. Notice the difference in size between the two.

An option to reduce the size of the Docker container is to start with a small base image. Usually, the base images for a distribution are fairly lean, but it might be a good idea to see what is installed in the image. If there are things that aren’t needed, you can then try creating your own base image that removes the unneeded tools.

Another option is to run the command `apt-get clean` to clean up any package caching that might be in the image.

Chapter 11. Scripts

11.1. DIGITS

11.1.1. run_digits.sh

```
#!/bin/bash
# file: run_digits.sh

mkdir -p $HOME/digits_workdir/jobs

cat <<EOF > $HOME/digits_workdir/digits_config_env.sh
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
EOF

docker run --gpus all --rm -ti --name=${USER}_digits -p 5000:5000 \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  --env-file=${HOME}/digits_workdir/digits_config_env.sh \
  -v /datasets:/digits_data:ro \
  --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
  nvcr.io/nvidia/digits:17.05
```

11.1.2. digits_config_env.sh

```
# DIGITS Configuration File
DIGITS_JOB_DIR=$HOME/digits_workdir/jobs
DIGITS_LOGFILE_FILENAME=$HOME/digits_workdir/digits.log
```

11.2. TensorFlow

11.2.1. run_tf_cifar10.sh

```
#!/bin/bash
# file: run_tf_cifar10.sh

# run example:
#  ./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# Get usage help via:
#  ./run_kerastf_cifar10.sh --help 2>/dev/null
```

```
_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
cifarcode=${_basedir}/examples/tensorflow/cifar/cifar10_multi_gpu_train.py
# cifarcode=${_basedir}/examples/tensorflow/cifar/cifar10_train.py

function join { local IFS="$1"; shift; echo "$*"; }

script_args=$(join : "$@")

dname=${USER}_tf

docker run --gpus all --name=$dname -d -t \
  --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
  -e cifarcode=$cifarcode -e script_args="$script_args" \
  nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname bash -c 'python $cifarcode ${script_args//:/ }'

docker stop $dname && docker rm $dname
```

11.3. Keras

11.3.1. venvfns.sh

```
#!/bin/bash
# file: venvfns.sh
# functions for virtualenv

[[ "${BASH_SOURCE[0]}" == "${0}" ]] && \
  echo Should be run as : source "${0}" && exit 1

enablevenvglobalsitepackages() {
 if ! [ -z ${VIRTUAL_ENV+x} ]; then
     _libpypath=$(dirname $(python -c \
  "from distutils.sysconfig import get_python_lib; print(get_python_lib())"))
    if ! [[ "${_libpypath}" == *"$VIRTUAL_ENV"* ]]; then
       return # VIRTUAL_ENV path not in the right place
    fi
       no_global_site_packages_file=${_libpypath}/no-global-site-packages.txt
    if [ -f $no_global_site_packages_file ]; then
        rm $no_global_site_packages_file;
        echo "Enabled global site-packages"
    else
        echo "Global site-packages already enabled"
    fi
 fi
}

disablevenvglobalsitepackages() {
 if ! [ -z ${VIRTUAL_ENV+x} ]; then
     _libpypath=$(dirname $(python -c \
  "from distutils.sysconfig import get_python_lib; print(get_python_lib())"))
    if ! [[ "${_libpypath}" == *"$VIRTUAL_ENV"* ]]; then
       return # VIRTUAL_ENV path not in the right place
    fi
    no_global_site_packages_file=${_libpypath}/no-global-site-packages.txt
    if ! [ -f $no_global_site_packages_file ]; then
```

```
        touch $no_global_site_packages_file
        echo "Disabled global site-packages"
    else
        echo "Global site-packages were already disabled"
    fi
 fi
}
```

11.3.2. setup_keras.sh

```
#!/bin/bash
# file: setup_keras.sh

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  nvcr.io/nvidia/cuda:8.0-cudnn6-devel-ubuntu16.04

docker exec -it -u root $dname \
  bash -c 'apt-get update && apt-get install -y virtualenv virtualenvwrapper'

docker exec -it $dname \
  bash -c 'source /usr/share/virtualenvwrapper/virtualenvwrapper.sh
  mkvirtualenv py-keras
  pip install --upgrade pip
  pip install keras --no-deps
  pip install PyYaml
  pip install numpy
  pip install scipy
  pip install ipython'

docker stop $dname && docker rm $dname
```

11.3.3. run_kerastf_mnist.sh

```
#!/bin/bash
# file: run_kerastf_mnist.sh

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
mnistcode=${_basedir}/examples/keras/mnist_cnn.py

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -w $workdir -e mnistcode=$mnistcode \
  nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
 bash -c 'source ~/.virtualenvs/py-keras/bin/activate
 source ~/venvfns.sh
 enablevenvglobalsitepackages
 python $mnistcode
 disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname
```

11.3.4. run_kerasth_mnist.sh

```
#!/bin/bash
# file: run_kerasth_mnist.sh

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
mnistcode=${_basedir}/examples/keras/mnist_cnn.py

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -w $workdir -e mnistcode=$mnistcode \
  nvcr.io/nvidia/theano:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
 bash -c 'source ~/.virtualenvs/py-keras/bin/activate
 source ~/venvfns.sh
 enablevenvglobalsitepackages
 KERAS_BACKEND=theano python $mnistcode
 disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname
```

11.3.5. run_kerastf_cifar10.sh

```
#!/bin/bash
# file: run_kerastf_cifar10.sh

# run example:
#  ./run_kerastf_cifar10.sh --epochs=3 --datadir=/datasets/cifar
# Get usage help via:
#  ./run_kerastf_cifar10.sh --help 2>/dev/null

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir
cifarcode=${_basedir}/examples/keras/cifar10_cnn_filesystem.py

function join { local IFS="$1"; shift; echo "$*"; }

script_args=$(join : "$@")

dname=${USER}_keras

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  -v /datasets/cifar:/datasets/cifar:ro -w $workdir \
  -e cifarcode=$cifarcode -e script_args="$script_args" \
  nvcr.io/nvidia/tensorflow:17.05

sleep 1 # wait for container to come up

docker exec -it $dname \
 bash -c 'source ~/.virtualenvs/py-keras/bin/activate
 source ~/venvfns.sh
 enablevenvglobalsitepackages
 python $cifarcode ${script_args//:/ }
```

```
 disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname
```

11.3.6. run_keras_script

```
#!/bin/bash
# file: run_keras_script.sh

_basedir="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"

# specify workdirectory for the container to run scripts or work from.
workdir=$_basedir

function join { local IFS="$1"; shift; echo "$*"; }

container="nvcr.io/nvidia/tensorflow:17.05"
backend="tensorflow"
script=''
datamnt=''

usage() {
cat <<EOF
Usage: $0 [-h|--help] [--container=container] [--script=script]
 [--<remain_args>]

 Sets up a keras environment. The keras environment is setup in a
 virtualenv and mapped into the docker container with a chosen
 --backend. Then runs the specified --script.

 --container - Specify desired container. Use "=" equal sign.
     Default: ${container}

 --backend - Specify the backend for Keras: tensorflow or theano.
     Default: ${backend}

 --script - Specify a script. Specify scripts with full or relative
     paths (relative to current working directory). Ex.:
            --script=examples/keras/cifar10_cnn_filesystem.py

 --datamnt - Data directory to mount into the container.

 --<remain_args> - Additional args to pass through to the script.

 -h|--help - Displays this help.

EOF
}

remain_args=()

while getopts ":h-" arg; do
 case "${arg}" in
 h ) usage
     exit 2
     ;;
 - ) [ $OPTIND -ge 1 ] && optind=$(expr $OPTIND - 1 ) || optind=$OPTIND
     eval _OPTION="\$$optind"
     OPTARG=$(echo $_OPTION | cut -d'=' -f2)
     OPTION=$(echo $_OPTION | cut -d'=' -f1)
     case $OPTION in
     --container ) larguments=yes; container="$OPTARG"  ;;
     --script ) larguments=yes; script="$OPTARG"  ;;
     --backend ) larguments=yes; backend="$OPTARG"  ;;
     --datamnt ) larguments=yes; datamnt="$OPTARG"  ;;
     --help ) usage; exit 2 ;;
```

```
     --* ) remain_args+=($_OPTION) ;;
     esac
    OPTIND=1
    shift
   ;;
 esac
done

script_args="$(join : ${remain_args[@]})"

dname=${USER}_keras

# formulate -v option for docker if datamnt is not empty.
mntdata=$([[ ! -z "${datamnt// }" ]] && echo "-v ${datamnt}:${datamnt}:ro" )

docker run --gpus all --name=$dname -d -t \
  -u $(id -u):$(id -g) -e HOME=$HOME -e USER=$USER -v $HOME:$HOME \
  $mntdata -w $workdir \
  -e backend=$backend -e script=$script -e script_args="$script_args" \
  $container

sleep 1 # wait for container to come up

docker exec -it $dname \
 bash -c 'source ~/.virtualenvs/py-keras/bin/activate
 source ~/venvfns.sh
 enablevenvglobalsitepackages
 KERAS_BACKEND=$backend python $script ${script_args//:/ }
 disablevenvglobalsitepackages'

docker stop $dname && docker rm $dname
```

11.3.7. cifar10_cnn_filesystem.py

```
#!/usr/bin/env python
# file: cifar10_cnn_filesystem.py
'''
Train a simple deep CNN on the CIFAR10 small images dataset.
'''

from __future__ import print_function
import sys
import os

from argparse import (ArgumentParser, SUPPRESS)
from textwrap import dedent


import numpy as np

# from keras.utils.data_utils import get_file
from keras.utils import to_categorical
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
import keras.layers as KL
from keras import backend as KB

from keras.optimizers import RMSprop


def parser_(desc):
 parser = ArgumentParser(description=dedent(desc))

 parser.add_argument('--epochs', type=int, default=200,
                     help='Number of epochs to run training for.')
```

```
 parser.add_argument('--aug', action='store_true', default=False,
                     help='Perform data augmentation on cifar10 set.\n')

 # parser.add_argument('--datadir', default='/mnt/datasets')
 parser.add_argument('--datadir', default=SUPPRESS,
                     help='Data directory with Cifar10 dataset.')

 args = parser.parse_args()

 return args


def make_model(inshape, num_classes):
 model = Sequential()
    model.add(KL.InputLayer(input_shape=inshape[1:]))
 model.add(KL.Conv2D(32, (3, 3), padding='same'))
 model.add(KL.Activation('relu'))
 model.add(KL.Conv2D(32, (3, 3)))
 model.add(KL.Activation('relu'))
 model.add(KL.MaxPooling2D(pool_size=(2, 2)))
 model.add(KL.Dropout(0.25))

 model.add(KL.Conv2D(64, (3, 3), padding='same'))
 model.add(KL.Activation('relu'))
 model.add(KL.Conv2D(64, (3, 3)))
 model.add(KL.Activation('relu'))
 model.add(KL.MaxPooling2D(pool_size=(2, 2)))
 model.add(KL.Dropout(0.25))

 model.add(KL.Flatten())
 model.add(KL.Dense(512))
 model.add(KL.Activation('relu'))
 model.add(KL.Dropout(0.5))
 model.add(KL.Dense(num_classes))
 model.add(KL.Activation('softmax'))

 return model


def cifar10_load_data(path):
 """Loads CIFAR10 dataset.

 # Returns
     Tuple of Numpy arrays: `(x_train, y_train), (x_test, y_test)`.
 """
 dirname = 'cifar-10-batches-py'
 # origin = 'http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz'
 # path = get_file(dirname, origin=origin, untar=True)
 path_ = os.path.join(path, dirname)

 num_train_samples = 50000

 x_train = np.zeros((num_train_samples, 3, 32, 32), dtype='uint8')
 y_train = np.zeros((num_train_samples,), dtype='uint8')

 for i in range(1, 6):
     fpath = os.path.join(path_, 'data_batch_' + str(i))
     data, labels = cifar10.load_batch(fpath)
     x_train[(i - 1) * 10000: i * 10000, :, :, :] = data
     y_train[(i - 1) * 10000: i * 10000] = labels

 fpath = os.path.join(path_, 'test_batch')
 x_test, y_test = cifar10.load_batch(fpath)

 y_train = np.reshape(y_train, (7, 1))
 y_test = np.reshape(y_test, (6, 1))
```

```
 if KB.image_data_format() == 'channels_last':
     x_train = x_train.transpose(0, 2, 3, 1)
     x_test = x_test.transpose(0, 2, 3, 1)

 return (x_train, y_train), (x_test, y_test)


def main(argv=None):
 '''
 '''
 main.__doc__ = __doc__
 argv = sys.argv if argv is None else sys.argv.extend(argv)
 desc = main.__doc__
 # CLI parser
 args = parser_(desc)

 batch_size = 32
 num_classes = 10
 epochs = args.epochs
 data_augmentation = args.aug

 datadir = getattr(args, 'datadir', None)

 # The data, shuffled and split between train and test sets:
 (x_train, y_train), (x_test, y_test) = cifar10_load_data(datadir) \
     if datadir is not None else cifar10.load_data()
 print(x_train.shape[0], 'train samples')
 print(x_test.shape[0], 'test samples')

 # Convert class vectors to binary class matrices.
 y_train = to_categorical(y_train, num_classes)
 y_test = to_categorical(y_test, num_classes)

 x_train = x_train.astype('float32')
 x_test = x_test.astype('float32')
 x_train /= 255
 x_test /= 255

 callbacks = None

 print(x_train.shape, 'train shape')
 model = make_model(x_train.shape, num_classes)

 print(model.summary())

 # initiate RMSprop optimizer
 opt = RMSprop(lr=0.0001, decay=1e-6)

 # Let's train the model using RMSprop
    model.compile(loss='categorical_crossentropy',
                optimizer=opt,
                metrics=['accuracy'])

 nsamples = x_train.shape[0]
 steps_per_epoch = nsamples // batch_size

 if not data_augmentation:
     print('Not using data augmentation.')
     model.fit(x_train, y_train,
                batch_size=batch_size,
                epochs=epochs,
                validation_data=(x_test, y_test),
                shuffle=True,
                callbacks=callbacks)

 else:
```

```
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        # set input mean to 0 over the dataset
        featurewise_center=False,
        samplewise_center=False,  # set each sample mean to 0
        # divide inputs by std of the dataset
           featurewise_std_normalization=False,
        # divide each input by its std
        samplewise_std_normalization=False,
        zca_whitening=False,  # apply ZCA whitening
        # randomly rotate images in the range (degrees, 0 to 180)
        rotation_range=0,
        # randomly shift images horizontally (fraction of total width)
        width_shift_range=0.1,
        # randomly shift images vertically (fraction of total height)
        height_shift_range=0.1,
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    model.fit_generator(datagen.flow(x_train, y_train,
                                     batch_size=batch_size),
                        steps_per_epoch=steps_per_epoch,
                       epochs=epochs,
                       validation_data=(x_test, y_test),
                        callbacks=callbacks)


if __name__ == '__main__':
 main()
```

Chapter 12. Troubleshooting

For more information about Docker containers, see:

- NGC User Guide
- NVIDIA-Docker GitHub
- HPC Visualization Containers User Guide

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

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com

Trademarks

NVIDIA, the NVIDIA logo, and cuBLAS, CUDA, DALI, DGX, DGX-1, DGX-2, DGX Station, DLProf, Jetson, Kepler, Maxwell, NCCL, Nsight Compute, Nsight Systems, NvCaffe, PerfWorks, Pascal, SDK Manager, Tegra, TensorRT, Triton Inference Server, Tesla, TF-TRT, and Volta are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
https://www.nvidia.com