Step-by-step guide to Docker

Basic commands

1. Run the hello-world Docker container to verify basic functionality

```
$ docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest:
sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://cloud.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

2. Pull an image from Docker Hub

```
docker pull <image name>
```

EXAMPLE:

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu
6b98dfc16071: Pull complete
4001a1209541: Pull complete
6319fc68c576: Pull complete
b24603670dc3: Pull complete
97f170c87c6f: Pull complete
Digest:
sha256:5f4bdc3467537cbbe563e80db2c3ec95d548a9145d64453b06939c4592d67b6d
```

```
Status: Downloaded newer image for ubuntu:latest
```

3. Run a container and print OS information

```
docker run [options] <image name> <command>
```

We can display information about the OS by printing the `/etc/os-release` file:

```
$ docker run ubuntu cat /etc/os-release

NAME="Ubuntu"
VERSION="18.04 LTS (Bionic Beaver)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 18.04 LTS"
VERSION_ID="18.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial
```

Compare this information with those from your native OS.

4. Run an interactive shell inside a container

```
docker run -it <image name> bash
```

`-i` stands for interactive

`-t` allocates a pseudo-TTY

EXAMPLE:

```
$ docker run -it ubuntu bash

root@e92829d2c1a4:/# whoami
  root
root@e92829d2c1a4:/# ls
  bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
  boot  etc  lib   media  opt  root  sbin  sys  usr
```

5. List Docker images in the system

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
ubuntu              latest              ebcd9d4fca80        3 days ago
118 MB
```

```
hello-world         latest              48b5124b2768        4 months ago
1.84 kB
```

6. Run a container from an image with a tag different from `latest`

   The general descriptor of Docker images on Docker Hub is in the form `<repository name>/<image name>:<image tag>`; in case of official repositories, the form is simply `<image name>:<tag>`. If the tag is not specified, Docker will default it to `latest`.

   EXAMPLE:

```
$ docker pull ubuntu:14.04

14.04: Pulling from library/ubuntu
cf0a75889057: Pull complete
c8de9902faf0: Pull complete
a3c0f7711c5e: Pull complete
e6391432e12c: Pull complete
624ce029a17f: Pull complete
Digest:
sha256:b2a55128abd84a99436157c2fc759cf0a525c273722460e6f8f9630747dfe7e8
Status: Downloaded newer image for ubuntu:14.04


$ docker run ubuntu:14.04 cat /etc/os-release

NAME="Ubuntu"
VERSION="14.04.5 LTS, Trusty Tahr"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 14.04.5 LTS"
VERSION_ID="14.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
```

   Compare this example with the one from point 2.

7. Write a simple Dockerfile

   Dockerfiles are made of a sequence of commands to incrementally build the environment that will constitute a Docker image. They are similar to Bash scripts, with the addition of some specific keywords, called instructions. Every statement in a Dockerfile must start with an instruction.

   The simplest and most useful instructions are:

   - **FROM**: identify an already existing image as a base image; the subsequent instructions in the Dockerfile will add stuff on top of what's already defined in that image.
   - **COPY**: copy files and directories from a source path into the image. The destination path will be automatically created if it does not exist.
   - **RUN**: execute any command as if you were into a shell. RUN instructions usually make up the most of a Dockerfile, either installing software from the package manager or downloading and compiling resources.
   - **ENV**: create a new environment variable in the image

EXAMPLE:

```
FROM debian:latest

RUN apt-get update && apt-get install -y wget

COPY script.sh /app

ENV NAME World
```

8. Build an image from a Dockerfile

```
docker build -t <name:tag> <Dockerfile path>
```

`-t` associates a user-supplied identifier to the new image. It is useful to already choose an identifier suitable for Docker Hub.

EXAMPLE (usually the Dockerfile is on the working directory):

```
$ docker build -t my_user/my_image:latest .
```

9. Login and push an image to Docker Hub

```
$ docker login

Login with your Docker ID to push and pull images from Docker Hub. If you
don't have a Docker ID,     head over to https://hub.docker.com to create one.
Username (<last logged user>):
Password:
Login Succeeded

$ docker push my_user/my_image:latest
```

Additional commands

1. List all Docker containers in the system (even stopped ones)

```
$ docker ps -a

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
e92829d2c1a4        ubuntu              "bash"                   About an hour
ago   Exited (0)   2 seconds ago                       reverent_shannon
45f97fea6fe8        ubuntu              "cat /etc/os-release"    2 hours ago
Exited (0)   2 hours ago                            silly_ride
1739aec2c30b        hello-world         "/hello"                 2 hours ago
Exited (0)   2 hours ago                            affectionate_euler
```

2. Run a container with automatic removal upon exit

   By default, Docker does not delete containers after they complete the tasks assigned to them and return control to the shell. Instead, those containers remain in a stopped state, ready to be resumed if the user wishes so. To run a container that will be automatically removed when it exits, use the `--rm` option of `docker run`.

   EXAMPLE:

```
$ docker ps -a

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
e92829d2c1a4        ubuntu              "bash"                   About an hour
ago   Exited (0) 2    seconds ago                       reverent_shannon
45f97fea6fe8        ubuntu              "cat /etc/os-release"    2 hours ago
Exited (0) 2    hours ago                         silly_ride
1739aec2c30b        hello-world         "/hello"                 2 hours ago
Exited (0) 2    hours ago                         affectionate_euler

$ docker run --rm ubuntu:16.04 cat /etc/os-release

NAME="Ubuntu"
VERSION="16.04.2 LTS (Xenial Xerus)"
ID=ubuntu
ID_LIKE=debian
PRETTY_NAME="Ubuntu 16.04.2 LTS"
VERSION_ID="16.04"
HOME_URL="http://www.ubuntu.com/"
SUPPORT_URL="http://help.ubuntu.com/"
BUG_REPORT_URL="http://bugs.launchpad.net/ubuntu/"
VERSION_CODENAME=xenial
UBUNTU_CODENAME=xenial

$ docker ps -a

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
e92829d2c1a4        ubuntu              "bash"                   About an hour
ago   Exited (0) 2    seconds ago                       reverent_shannon
45f97fea6fe8        ubuntu              "cat /etc/os-release"    2 hours ago
Exited (0) 2    hours ago                         silly_ride
1739aec2c30b        hello-world         "/hello"                 2 hours ago
Exited (0) 2    hours ago                         affectionate_euler
```

3. Remove containers

```
docker rm <container ID or name> [<container ID or name>...]
```

EXAMPLE:

```
$ docker ps -a

CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
e92829d2c1a4        ubuntu              "bash"                   About an hour
ago   Exited (0) 2 seconds ago                       reverent_shannon
45f97fea6fe8        ubuntu              "cat /etc/os-release"    2 hours ago
Exited (0) 2 hours ago                              silly_ride
1739aec2c30b        hello-world         "/hello"                 2 hours ago
Exited (0) 2 hours ago                              affectionate_euler

$ docker rm e92829d2c1a4

e92829d2c1a4

$ docker rm silly_ride 1739aec2c30b

silly_ride
1739aec2c30b

$ docker ps -a

CONTAINER ID     IMAGE     COMMAND      CREATED     STATUS       PORTS
NAMES
```

A useful combination to remove **all** containers on the system with a single command is the following:

```
$ docker rm $(docker ps -aq)
```

4. Remove Docker images

```
docker rmi <image ID or name> [<image ID or name>...]
```

EXAMPLE:

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
ubuntu              latest              ebcd9d4fca80        3 days ago
118 MB
hello-world         latest              48b5124b2768        4 months ago
1.84 kB


$ docker rmi ebcd9d4fca80 hello-world

Untagged: ubuntu:latest
Untagged:
ubuntu@sha256:382452f82a8bbd34443b2c727650af46aced0f94a44463c62a9848133ecb1aa8
Deleted:
sha256:ebcd9d4fca80e9e8afc525d8a38e7c56825dfb4a220ed77156f9fb13b14d4ab7
```

```
Deleted:
sha256:ef5b99eed7c2ed19ef39f72ac19bb66e16ed6c0868053daae60306a73858fbd4
Deleted:
sha256:257e51479af1e9d2e0c9b958e68f6b992329904df24d81efa191cef515a9bf8b
Deleted:
sha256:6e1d2d371500e2fe6df75f5755d0b9f2a3b69a42fe88100d514212bbba7ad23f
Deleted:
sha256:afa9e7a5e3f3b006942d128c562a3273947c7ab50cdac33fea7213890072a5b6
Deleted:
sha256:2df9b8def18a090592bf1cbd1079e1ac2274435c53f027ee5ce0a8faaa5d6d4b
Untagged: hello-world:latest
Untagged: hello-
world@sha256:c5515758d4c5e1e838e9cd307f6c6a0d620b5e07e6f927b07d05f6d12a1ac8d7
Deleted:
sha256:48b5124b2768d2b917edcb640435044a97967015485e812545546cbed5cf0233
Deleted:
sha256:98c944e98de8d35097100ff70a31083ec57704be0991a92c51700465e4544d08

$ docker images

REPOSITORY      TAG      IMAGE ID      CREATED     SIZE
```

5. Assign a different identifier to an existing image

```
docker tag <source image> <target image>
```

EXAMPLE:

```
$ docker tag dummy_image my_user/awesome_image:latest
```

Additional Dockerfile instructions

- **ADD**: copy files, directories and remote file URLs from a source into the image. It also automatically extracts tar archives into the image, which constitutes its best use case. Since the additional features of ADD are not immediately obvious, the official Docker documentation indicates COPY as the preferred instruction if files have to simply be transferred into an image.
- **LABEL**: add metadata to an image in a key-value pair. An image can have multiple labels. Labels are additive, including labels in the base image indicated with FROM. Labels are useful for improved image classification and are sometimes used by third-party software.
- **WORKDIR**: set the working directory for subsequent instructions in the Dockerfile. If the WORKDIR doesn't exist, it will be created. Without a WORKDIR instruction, all actions in a Docker file happen at the filesystem root.

Basic Dockerfile good practices

- **Do not use too many image layers:** Docker images are built from a series of layers, stacked on top of each other. Each layer represents an instruction in the image's Dockerfile and is simply a set of differences from the layer before it.

  Try to achieve a balance between readability of the Dockerfile and reducing the number of image layers. Minimizing the number of layers also benefits total image size and performance of build and pull processes.

- **Cleanup after installations:** Because of the layered structure of Docker images, if resources are downloaded and removed with different instructions, a copy of those resources will still exist in the layer associated with the first instruction that retrieved them.

  You can reduce the total image size by using a single RUN instruction that also cleans the package manager cache, or performs a complete installation from source and removes the original code.

  Examples:

```
# Install from package manager and clean its cache
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
           build-essential \
           wget \
    && rm -rf /var/lib/apt/lists/*

# Install from source and remove the code
RUN wget -q http://www.something.org/source-package.tar.gz \
    && tar xf source-package.tar.gz \
    && cd source-package \
    && ./configure \
    && make \
    && make install \
    && cd .. \
    && rm -rf source-package \
    && rm source-package.tar.gz
```

- **Avoid invalidating the build cache:** Each time `docker build` executes a Dockerfile instruction successfully, it caches the resulting image (even if it is an intermediate layer). When carrying out future builds of the Dockerfile, Docker will look for a match of a given instruction in its cache and, if found, it will reuse it instead of re-building the layer. Thus, proper use of the build cache can greatly speed up the creation of images.

  Generally, if an instruction changes in a previously built Dockerfile, the lookup will fail and the build cache will be invalidated. When this happens, all subsequent Dockerfile commands will re-build new layers and the cache will not be used.

  Special care should be used with `ADD` and `COPY` instructions: the contents of the files in the images are checksummed and, during cache lookup, the new checksum is compared against the checksum in the cached images. If anything has changed in the file(s), such as the contents and metadata, then the cache is invalidated. This means that even if the Dockerfile is identical, but you changed the files copied by an `ADD` or `COPY` instruction, a full rebuild will happen from that instruction onwards.

Running GPU-accelerated containers with nvidia-docker (Linux only)

`nvidia-docker` will provide Docker with a custom runtime (called NVIDIA Container Runtime) that is able to expose NVIDIA GPUs and the CUDA driver inside containers. It is activated by passing the `--runtime=nvidia` option to `docker run`.

1. Detect native GPUs from the NVIDIA CUDA base container:

```
$ docker run --runtime=nvidia --rm nvidia/cuda nvidia-smi

Using default tag: latest
latest: Pulling from nvidia/cuda
```

```
b6f892c0043b: Pull complete
55010f332b04: Pull complete
2955fb827c94: Pull complete
3deef3fcbd30: Pull complete
cf9722e506aa: Pull complete
b781c303fe6d: Pull complete
6478274efc14: Pull complete
c977ea0ae412: Pull complete
6be681c439f5: Pull complete
0911dbb0ec93: Pull complete
Digest:
sha256:bad7cb6adc819942226e80eeb90f79fc7636f453d27f3d2ad47163ff15ac13fe
Status: Downloaded newer image for nvidia/cuda:latest
Fri May 19 15:58:43 2017
+-----------------------------------------------------------------------------
+
| NVIDIA-SMI 375.26                 Driver Version: 375.26
|
|-------------------------------+----------------------+----------------------
+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC
|
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M.
|
|===============================+======================+======================
|
|   0  Quadro K1100M       Off  | 0000:01:00.0     Off |                  N/A
|
| N/A   44C    P8    N/A /  N/A |      0MiB /  1998MiB |      0%      Default
|
+-------------------------------+----------------------+----------------------
+

+-----------------------------------------------------------------------------
+
| Processes:                                                       GPU Memory
|
|  GPU       PID  Type  Process name                               Usage
|
|=============================================================================
|
|  No running processes found
|
+-----------------------------------------------------------------------------
+
```

2. Use the `deviceQuery` sample from the CUDA SDK to print GPU-related information from a container.

   We have already built an image with compiled CUDA samples, and you can retrieve it from Docker Hub using the descriptor `ethcscs/cudasamples:8.0`.

```
$ docker run --runtime=nvidia --rm ethcscs/cudasamples:8.0
/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery

/usr/local/cuda/samples/1_Utilities/deviceQuery/deviceQuery Starting...

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

Detected 1 CUDA Capable device(s)
```

```
Device 0: "Quadro K1100M"
  CUDA Driver Version / Runtime Version          8.0 / 8.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 1998 MBytes (2095251456
bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            706 MHz (0.71 GHz)
  Memory Clock rate:                             1400 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536,
65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048
layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device
simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 8.0, CUDA Runtime
Version = 8.0, NumDevs = 1, Device0 =     Quadro K1100M
Result = PASS
```

3. Verify that container applications can actually benefit from GPU acceleration. Here we run the CUDA SDK N-body sample, first using the GPU and then the CPU for computations:

```
$ docker run --runtime=nvidia --rm ethcscs/cudasamples:8.0
/usr/local/cuda/samples/5_Simulations/nbody/nbody -benchmark -numbodies=2048

[... N-body sample preamble ...]

> Windowed mode
> Simulation data stored in video memory
> Single precision floating point simulation
> 1 Devices used for simulation
GPU Device 0: "Quadro K1100M" with compute capability 3.0

> Compute 3.0 CUDA device: [Quadro K1100M]
number of bodies = 2048
2048 bodies, total time for 10 iterations: 4.035 ms
= 10.394 billion interactions per second
= 207.874 single-precision GFLOP/s at 20 flops per interaction
```

```
$ nvidia-docker run --rm ethcscs/dockerfiles:cudasamples8.0
/usr/local/cuda/samples/5_Simulations/nbody/nbody -benchmark -numbodies=2048 -
cpu

[... N-body sample preamble ...]

> Windowed mode
> Simulation data stored in video memory
> Single precision floating point simulation
> 1 Devices used for simulation
> Simulation with CPU
number of bodies = 2048
2048 bodies, total time for 10 iterations: 1025.307 ms
= 0.041 billion interactions per second
= 0.818 single-precision GFLOP/s at 20 flops per interaction
```

Notice that with the GPU (on a notebook) the code achieves 207.9 GFLOP/s, while with the CPU it only manages 0.8 GFLOP/s.

Advanced topic: Creating custom CUDA images

The base images provided by NVIDIA under the `nvidia/cuda` Docker Hub repository only offer flavors based on Ubuntu and CentOS. If you want to build a CUDA-enabled image on a different distribution, we offer the following advice:

Installing the CUDA Toolkit

- **Package manager installer:** repository installers (to be used through the system package manager) are available for Fedora, OpenSUSE, RHEL and SLES (and also for Ubuntu and CentOS, if you don't want to use NVIDIA's images). For detailed installation instructions, refer to the official CUDA Toolkit Documentation (http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#package-manager-installation).

  Please note that installing the default package options (e.g. `cuda` or `cuda-tookit`) will add a significant amount of resources to your image (more than 1GB). Significant size savings can be achieved by selectively installing only the packages with the parts of the Toolkit that you need. Please consult your system package manager documentation in order to list the available packages inside the CUDA repositories.

- **Runfile install:** some distributions are not covered by the CUDA package manager and have to perform the installation through the standalone runfile installer. One such case is Debian, which is also used as base for several official Docker Hub images (e.g. Python). For detailed installation instructions, refer to the official CUDA Toolkit Documentation (http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html#runfile). We advise to supply the `--silent` and `--toolkit` options to the installer, to avoid installing the CUDA drivers as well.

  The standalone installer will add a significant amount of resources to your image, including documentation and SDK samples. If you are really determined to reduce the size of your image, you can selectively `rm -rf` the parts of the Toolkit that you don't need, but be careful about not deleting libraries and tools that may be used by applications in the container!

Controlling the NVIDIA Container Runtime

The NVIDIA Container Runtime is controlled through several environment variables. These can be part of the container image or be set by `docker run` with the `-e`, `--env` or `--env-file` options (please refer to the official documentation for the full syntax of these flags). A brief description of the most useful variables follows:

- `NVIDIA_VISIBLE_DEVICES`: This variable controls which GPUs will be made accessible inside the container. The values can be a list of indexes (e.g. `0,1,2`), `all`, or `none`. If the variable is empty or unset, the runtime will behave as default Docker.

- `NVIDIA_DRIVER_CAPABILITIES`: This option controls which driver libraries and binaries will be mounted inside the container. It accepts as values a comma-separated list of driver features (e.g. `utility,compute`) or the string `all` (for all driver capabilities). An empty or unset variable will default to minimal driver capabilties (`utility`).

- `NVIDIA_REQUIRE_CUDA`: A logical expression to define a constraint on the CUDA driver version required by the container (e.g. `"cuda>=8.0"` will require a driver supporting the CUDA Toolkit version 8.0 and later). Multiple constraints can be expressed in a single environment variable: space-separated constraints are ORed, comma-separated constraints are ANDed.

A full list of the environment variables looked up by the NVIDIA Container Runtime and their possible values is available here.

As an example, in order to work correctly with the NVIDIA runtime, an image including a CUDA 8.0 application could

feature these instructions in its Dockerfile:

```
ENV NVIDIA_VISIBLE_DEVICES all
ENV NVIDIA_DRIVER_CAPABILITIES compute,utility
ENV NVIDIA_REQUIRE_CUDA "cuda>=8.0"
```

Alternatively, the variables can be specified (or even overwritten) from the command line:

```
$ docker run --runtime=nvidia -e NVIDIA_VISIBLE_DEVICES=all -e
NVIDIA_DRIVER_CAPABILITIES=compute,utility --rm nvidia/cuda nvidia-smi
```

The official `nvidia/cuda` images already include a set of these environment variables. This means that all the Dockerfiles that do a `FROM nvidia/cuda` will automatically inherit them and thus will work seamlessly with the NVIDIA Container Runtime.