USING THE NVIDIA HPC SDK WITH CONTAINERS

DU-09866-001-V2023 | November 2023

TABLE OF CONTENTS

Chapter 1. INTRODUCTION

Welcome to the HPC SDK Container Documentation.

This book is designed to provide you with information on NVIDIA's HPC application containers.

1.1. Introduction to Containers

Containers are a lighter weight virtualization technology based on Linux namespaces. Unlike virtual machines, containers share the kernel and other services with the host. As a result, containers can startup very quickly and have negligible performance overhead, but they do not provide the full isolation of virtual machines.

Containers bundle the entire application user space environment into a single image. This way, the application environment is both portable and consistent, and agnostic to the underlying host system software configuration. Container images can be deployed widely, and even shared with others, with confidence that the results will be reproducible.

Containers make life simpler for developers, users, and system administrators. Developers can distribute software in a container to provide a consistent runtime environment and reduce support overhead. Container images from repositories such as NGC can help users start up quickly on any system and avoid the complexities of building from source. Containers can also help IT staff tame environment module complexity and support legacy workloads that are no longer compatible with host operating system.

There are many container runtimes; two of the most significant are Docker and Singularity, both of which are covered in this guide.

Chapter 2. NGC

NVIDIA HPC SDK containers are available on NGC and are the best way to get started using the HPC SDK and containers. Two types of containers are provided, "devel" containers which contain the entire HPC SDK development environment, and "runtime" container which include only the components necessary to redistribute software built with the HPC SDK.

2.1. Using the NGC Containers with Docker

Several source code examples are available in the "devel" container at `/opt/nvidia/hpc_sdk/$NVARCH/23.11/examples`, where **`$NVARCH`** is either **`Linux_x86_64`** or **`Linux_aarch64`** depending on the system architecture.

To access the OpenACC examples in an interactive session, use:

```
$ sudo docker run --gpus all -it --rm nvcr.io/nvidia/nvhpc:23.11-devel-
cuda_multi-ubuntu22.04
# cd /opt/nvidia/hpc_sdk/Linux_x86_64/23.11/examples/OpenACC/samples
# make all
```

For your own code, the following command mounts the current directory on the host as `/src` inside the container and starts an interactive session as the current user inside the container.

```
$ sudo docker run --gpus all -it --rm --user $(id -u):$(id -g) --volume $(pwd):/
src --workdir /src nvcr.io/nvidia/nvhpc:23.11-devel-cuda_multi-ubuntu22.04
```

Since containers are ephemeral, this command mounts the source code directory from the host as **`/src`** inside the container (**`--volume`**) and defaults to this directory when the container starts (**`--workdir`**). Any changes to the source code or build artifacts made from inside the container will be stored in the source directory on the host and persist even when the container exits.

By default, the user inside a Docker container is **`root`**. The **`--user`** option modifies this so the user inside the container is the same as the user outside the container. Without

this option the build artifacts and other files created inside the container in the **/src** directory would be owned by **root**.

The other options tell Docker to cleanup the container when the container exits (**--rm**), to enable NVIDIA GPUs (**--gpus all**), and that this is an interactive session (**-it**).

Assuming there is a Makefile in the /src directory, then the make command can be appended to build the source using the HPC SDK container with the build artifacts available in the current directory on the host.

```
$ sudo docker run --gpus all -it --rm --user $(id -u):$(id -g) --volume $(pwd):/
src --workdir /src nvcr.io/nvidia/nvhpc:23.11-devel-cuda_multi-ubuntu22.04 make
```

2.2. Using the NGC Containers with Singularity

When using Singularity, typically the container image is saved locally as a Singularity Image File (SIF). This command saves the container in the current directory as nvhpc-23.11-devel.sif.

```
$ singularity build nvhpc-23.11-devel.sif docker://nvcr.io/nvidia/nvhpc:23.11-
devel-cuda_multi-ubuntu22.04
```

The following command starts an interactive session as the current user inside the container.

```
$ singularity shell --nv nvhpc-23.11-devel.sif
```

Unlike Docker the user inside a Singularity container is the same as the user outside the container and the user's home directory, current directory, and **/tmp** are automatically mounted inside the container.

The only additional option needed is **--nv** to enable NVIDIA GPU support.

Assuming there is a Makefile in the current directory, then the make command can be appended to build the source using the HPC SDK container with the build artifacts available in the current directory on the host.

```
$ singularity exec --nv nvhpc-23.11-devel.sif make
```

Chapter 3. BUILDING CONTAINERIZED APPLICATIONS

When the interactive development cycle is complete, a container is an excellent way to broadly distribute the result. Container images are generated from container specification files

- Dockerfiles for Docker and other container builders,
- Singularity definition files for Singularity.

The set of instructions are mostly the same for Docker and Singularity, but the syntax is different.

To minimize the container image size and adhere to the permissible redistribution of the HPC SDK, only the application itself and its runtime dependencies should be included in the container. Docker and Singularity both support multi-stage container builds. A multi-stage container specification typically consists of 2 parts:

1. A build stage based on a full development environmentand application source code, and
2. A distribution stage based on a smaller runtime environment that cherry picks content from the build stage such as the application binary and other build artifacts.

The CloverLeaf mini application is used to illustrate these concepts.

3.1. Building Containerized Applications with Docker

To build a container image with Docker:

```
$ sudo docker build -t <name:tag> -f Dockerfile .
```

The **`-t`** option specifies the name to give the container image, in the **`name:tag`** format.

The build stage of the CloverLeaf Dockerfile is based on the HPC SDK development image from NGC. The runtime stage is based on the smaller HPC SDK runtime image,

also from NGC. HPC SDK runtime images are available for each CUDA version bundled with the HPC SDK; select the version corresponding to the CUDA version used to build CloverLeaf. The **clover_leaf** binary and the sample input datasets are copied from the build stage.

Finally, the directory **/opt/CloverLeaf-OpenACC/bin** is added to the default **PATH** for convenience.

```
# Build stage
FROM nvcr.io/nvidia/nvhpc:23.11-devel-cuda_multi-ubuntu22.04 AS build

# build CloverLeaf
RUN mkdir /source && \
    cd /source && \
    git clone https://github.com/UoB-HPC/CloverLeaf-OpenACC.git && \
    cd CloverLeaf-OpenACC && \
    make COMPILER=PGI FLAGS_PGI="-Mpreprocess -fast -acc -Minfo=acc -gpu=ccall -
tp=px"

# Runtime stage
FROM nvcr.io/nvidia/nvhpc:23.11-runtime-cuda11.8-ubuntu22.04

COPY --from=build /source/CloverLeaf-OpenACC/clover_leaf /opt/CloverLeaf-
OpenACC/bin/
COPY --from=build /source/CloverLeaf-OpenACC/InputDecks /opt/CloverLeaf-OpenACC/
InputDecks

ENV PATH=/opt/CloverLeaf-OpenACC/bin:$PATH
```

To generate the container image from this Dockerfile:

```
$ sudo docker build -t cloverleaf:git -f Dockerfile .
```

The containerized CloverLeaf can be run using the following command:

```
$ sudo docker run --gpus all --cap-add=SYS_NICE --rm cloverleaf:git mpirun -n 1
 --allow-run-as-root clover_leaf
```

--cap-add=SYS_NICE is required to allow MPI to set the CPU affinity.

This will run the small built-in dataset; to use one of the sample datasets mount it from the host into the working directory (**/**) as **clover.in**.

The dataset should be typically mounted from the host into the running container. Including datasets in the container image is bad practice and is not recommended. Datasets can be large and bloat the size of the container image and are often specific to a particular usage. However, neither of these conditions are true for CloverLeaf: the CloverLeaf input files are tiny and they are standard.

```
$ sudo docker run --gpus all --rm --volume $(pwd)/clover_bm32_short.in:/
clover.in cloverleaf:git mpirun -n 1 --allow-run-as-root clover_leaf
```

3.2. Building Containerized Applications with Singularity

To build a container image with Singularity:

```
$ sudo singularity build <image> Singularity.def
```

<image> is the name of the resulting Singularity image file (SIF).

The instructions for building the CloverLeaf container are virtually identical to the corresponding Dockerfile in the previous section, although the syntax differs. The key difference is the step to configure the container environment; when building containers Singularity does not automatically setup the environment inherited from Docker base images and this must be done manually.

```
# Build stage
BootStrap: docker
From: nvcr.io/nvidia/nvhpc:23.11-devel-cuda_multi-ubuntu22.04
Stage: build

%post
  . /.singularity.d/env/10-docker*.sh

  # build CloverLeaf
  mkdir /source
  cd /source
  git clone https://github.com/UoB-HPC/CloverLeaf-OpenACC.git
  cd CloverLeaf-OpenACC
  make COMPILER=PGI FLAGS_PGI="-Mpreprocess -fast -acc -Minfo=acc -gpu=ccall -
tp=px"

# Runtime stage
BootStrap: docker
From: nvcr.io/nvidia/nvhpc:23.11-runtime-cuda11.8-ubuntu22.04

%files from build
  /source/CloverLeaf-OpenACC/clover_leaf /opt/CloverLeaf-OpenACC/bin/clover_leaf
  /source/CloverLeaf-OpenACC/InputDecks  /opt/CloverLeaf-OpenACC/InputDecks

%environment
  export PATH=/opt/CloverLeaf-OpenACC/bin:$PATH
```

To generate the container image from this Singularity definition file:

```
$ sudo singularity build cloverleaf-git.sif Singularity.def
```

The containerized CloverLeaf can be run using the following command:

```
$ singularity run --nv cloverleaf-git.sif mpirun -n 1 clover_leaf
```

This will run the small built-in dataset; to use one of the sample datasets copy it to the current working directory on the host as clover.in.

```
$ cp CloverLeaf-OpenACC/InputDecks/clover_bm32_short.in clover.in
$ singularity run --nv cloverleaf-<label>.sif mpirun -n 1 clover_leaf
```

3.3. HPC Container Maker

CloverLeaf, like nearly all mini-apps, is intentionally simple and has a very limited set of build dependencies, essentially a compiler and an MPI library. Real world applications are typically much more complex and may have dependencies on third-party software components. As a result, containerizing real world application may require considerably more effort than a mini-app like CloverLeaf.

HPC Container Maker (HPCCM) is an open source tool to make it easier to generate container specification files. HPCCM generates Dockerfiles or Singularity definition files from a high level Python recipe. HPCCM recipes have some distinct advantages over "native" container specification formats.

1. A library of HPC building blocks that separate the choice of what to include in a container image from the details of how it's done. The building blocks transparently provide the latest component and container best practices.
2. Python provides increased flexibility over static container specification formats. Python-based recipes can branch, validate user input, etc. - the same recipe can generate multiple container specifications.
3. Generate either Dockerfiles or Singularity definition files from the same recipe.

The following is the HPCCM Python recipe for the scenario where CloverLeaf is checked out from the GitHub repository:

```
# Build stage
Stage0 += baseimage(image='nvcr.io/nvidia/nvhpc:23.11-devel-cuda_multi-
ubuntu22.04', _as='build')

# build CloverLeaf
Stage0 += generic_build(build=['make COMPILER=PGI FLAGS_PGI="-Mpreprocess -fast
 -acc -Minfo=acc -gpu=ccall -tp=px"'],
    install=['install -m 755 -d /opt/CloverLeaf-OpenACC/bin',
      'install -m 755 clover_leaf /opt/CloverLeaf-OpenACC/bin/clover_leaf',
      'install -m 755 -d /opt/CloverLeaf-OpenACC/InputDecks',
      'install -m 644 InputDecks/* /opt/CloverLeaf-OpenACC/InputDecks'],
    prefix='/opt/CloverLeaf-OpenACC',
    repository='https://github.com/UoB-HPC/CloverLeaf-OpenACC.git')

# Runtime stage
Stage1 += baseimage(image='nvcr.io/nvidia/nvhpc:23.11-runtime-cuda11.8-
ubuntu22.04')

Stage1 += Stage0.runtime()
Stage1 += environment(variables={'PATH': '/opt/CloverLeaf-OpenACC/bin:$PATH'})
```

The HPCCM recipe is mostly descriptive, rather than prescriptive like the Dockerfile and Singularity definition files. For instance, the recipe specifies the CloverLeaf git repository, but does not define the details of how it should be downloaded. Since the CloverLeaf Makefile does not provide an install target, a basic install method needs to be specified. HPCCM also includes generic_autotools and **`generic_cmake`** building blocks for packages that use GNU **`Autotools`** or **`CMake`**, respectively.

The **hpccm** command line tool processes the recipe and outputs either a Dockerfile or a Singularity definition file. The resulting container specification file should be used as shown in the above sections to generate a container image.

```
$ hpccm --recipe cloverleaf.py

$ hpccm --recipe cloverleaf.py --format singularity --singularity-version 3.2
```

Multi-stage container builds were added to Singularity in version 3.2. However, the multi-stage Singularity definition file syntax is incompatible with earlier Singularity versions. The HPCCM **--singularity-version** flag is currently necessary to generate multi-stage Singularity definition files; otherwise the output is a single stage Singularity definition file compatible with all versions.

HPCCM includes building blocks for many common HPC software components such as HDF5, FFTW, OpenMPI, and many more that may be required for real world applications. One of the building blocks covers the HPC SDK, making it easy to include in custom images.

Chapter 4. BEST PRACTICES

The following sections discuss some of the best practices when using HPC SDK Containers.

4.1. Multi Stage Builds

Multi-stage builds are a way to control the size of container images. In the same Dockerfile, you can define a second stage that is a completely separate container image and copy just the binary and any runtime dependencies from preceding stages into the image. The output of a multi-stage build is a single container image corresponding to the last stage of the Dockerfile. This method can also be used to prevent redistribution of source code. Multi-stage builds have been used extensively in the preceding sections.

4.2. Layering

The OCI image format used by Docker is layered. OCI container images are composed of a series of layers. The layers are applied sequentially, one on top of another, to form the container image that you ultimately see when running a container.

The layers are cached and the Docker container builder can take advantage of the layer cache to speed up builds of container with common layers. Also, builds that were interrupted can be resumed from the previous point in the cache rather than having to start over from scratch.

However, care must be taken when specifying the Dockerfile instructions not to inadvertently bloat the container image size. The OCI image specification employs file level deduplication to handle conflicts. When a build instruction creates or modifies a file, the entire file is saved in the corresponding layer. For example, the following Dockerfile instructions generate seven (7) separate layers.

```
RUN mkdir /source
RUN cd /source && git clone https://github.com/UoB-HPC/CloverLeaf-OpenACC.git
RUN cd /source/CloverLeaf-OpenACC && make COMPILER=PGI FLAGS_PGI="-Mpreprocess -
fast -acc -Minfo=acc -gpu=ccall -tp=px"
RUN mkdir -p /opt/CloverLeaf-OpenACC
RUN install -m 755 -d /opt/CloverLeaf-OpenACC/bin
```

```
RUN install -m 755 clover_leaf /opt/CloverLeaf-OpenACC/bin/clover_leaf
RUN install -m 755 -d /opt/CloverLeaf-OpenACC/InputDecks
RUN install -m 644 InputDecks/* /opt/CloverLeaf-OpenACC/InputDecks
RUN rm -rf /source/CloverLeaf-OpenACC
```

Despite the final instruction to remove the git checkout of the CloverLeaf source code, the source code is still actually present in the image - the final layer is just a whiteout file entry.

A best practice arising from file level deduplication of layers is to put all actions modifying the same set of files in the same Dockerfile instruction. For example, remove any temporary files in the same instruction in which they are created. Generally, put all instructions relating to the same component in the same layer, but use separate layers for different components. HPCCM automatically generates container specification files that follow this best practice.

4.3. Multi Architecture Support

One of the goals of containerizing an application is to allow it to run on a wide variety of systems, including across different CPU and GPU generations. However, the CloverLeaf **`Makefile`** sets the GPU compute capability to cc60, i.e., Pascal. The default CloverLeaf binary may not run on older or newer GPUs. Similarly, the NVIDIA compiler implicitly optimizes for the build system CPU architecture. If the container was built on a system with the latest CPU microarchitecture, it would not run on systems with older CPUs, or conversely if built on a system with an older CPU microarchitecture, it may not run optimally on newer CPUs.

An approach to deal with the tension between supporting a wide range of systems and delivering optimal performance is to build *fat* or *unified* binaries. Support for multiple compute capabilities and instruction sets can be embedded in a single binary and the optimal code path will be used automatically at runtime. This is the approach taken by specifying the **`-gpu=ccall`** compiler option, which will generate code for multiple GPU compute capabilities in the same binary.

Another approach to build for the "lowest common denominator". This is the approach taken for the CPU microarchitecture by specifying the **`-tp=px`** compiler option.

Unfortunately the build systems of some real world applications do not allow the unified binary approach. An alternative is to build and redistribute multiple binaries in the same container image. A container entry point can be used to detect the system architecture at runtime and setup the environment (**`PATH`**, **`LD_LIBRARY_PATH`**, etc.) inside the container accordingly. Since HPCCM recipes are Python, building multiple binaries is as simple as a **`for`** loop. For example, to build for multiple CPU architectures:

```
for cpu_arch in ['sandybridge', 'haswell', 'skylake', 'icelake']:
  Stage0 += generic_build(build=['make COMPILER=PGI FLAGS_PGI="-Mpreprocess -
fast -acc -Minfo=acc -ta=tesla,ccall -tp={}"'.format(cpu_arch)],
        install=['install -m 755 -d /opt/CloverLeaf-OpenACC-{}/
bin'.format(cpu_arch),
            'install -m 755 clover_leaf /opt/CloverLeaf-OpenACC-{}/
bin'.format(cpu_arch)],
        prefix='/opt/CloverLeaf-OpenACC-{}'.format(cpu_arch),
        repository='https://github.com/UoB-HPC/CloverLeaf-OpenACC.git')
```

In general, the same recipe can also be used for different processor architectures as well. The HPC SDK NGC containers are provided for both Arm and X86 processors. With a slight tweak to the **`-tp px`** compiler option, the provided Dockerfile, Singularity definition file, and HPCCM recipe can also be used to generate Arm container images.

4.4. Version Tagging and Reproducibility

Container specification files often download content from online repositories such as GitHub. That content can change from one point in time to another. For instance, the master branch of the CloverLeaf GitHub repository could change in a way that invalidates the container specification. To avoid this scenario, specify a tag or commit to ensure that the container image build is reproducible

A HPCCM recipe can checkout a specific commit or tag to increase the reproducibility of the recipe.

```
 Stage0 += generic_build(build=['make COMPILER=PGI FLAGS_PGI="-Mpreprocess -fast
 -acc -Minfo=acc -gpu=ccall -tp=px"'],
        commit='23b8e81b5234474757e44418e78ce91c8d050363',
        install=['install -m 755 -d /opt/CloverLeaf-OpenACC/bin',
            'install -m 755 clover_leaf /opt/CloverLeaf-OpenACC/bin/
clover_leaf',
            'install -m 755 -d /opt/CloverLeaf-OpenACC/InputDecks',
            'install -m 644 InputDecks/* /opt/CloverLeaf-OpenACC/InputDecks'],
        prefix='/opt/CloverLeaf-OpenACC',
        repository='https://github.com/UoB-HPC/CloverLeaf-OpenACC.git')
```

Notice

Trademarks

Copyright