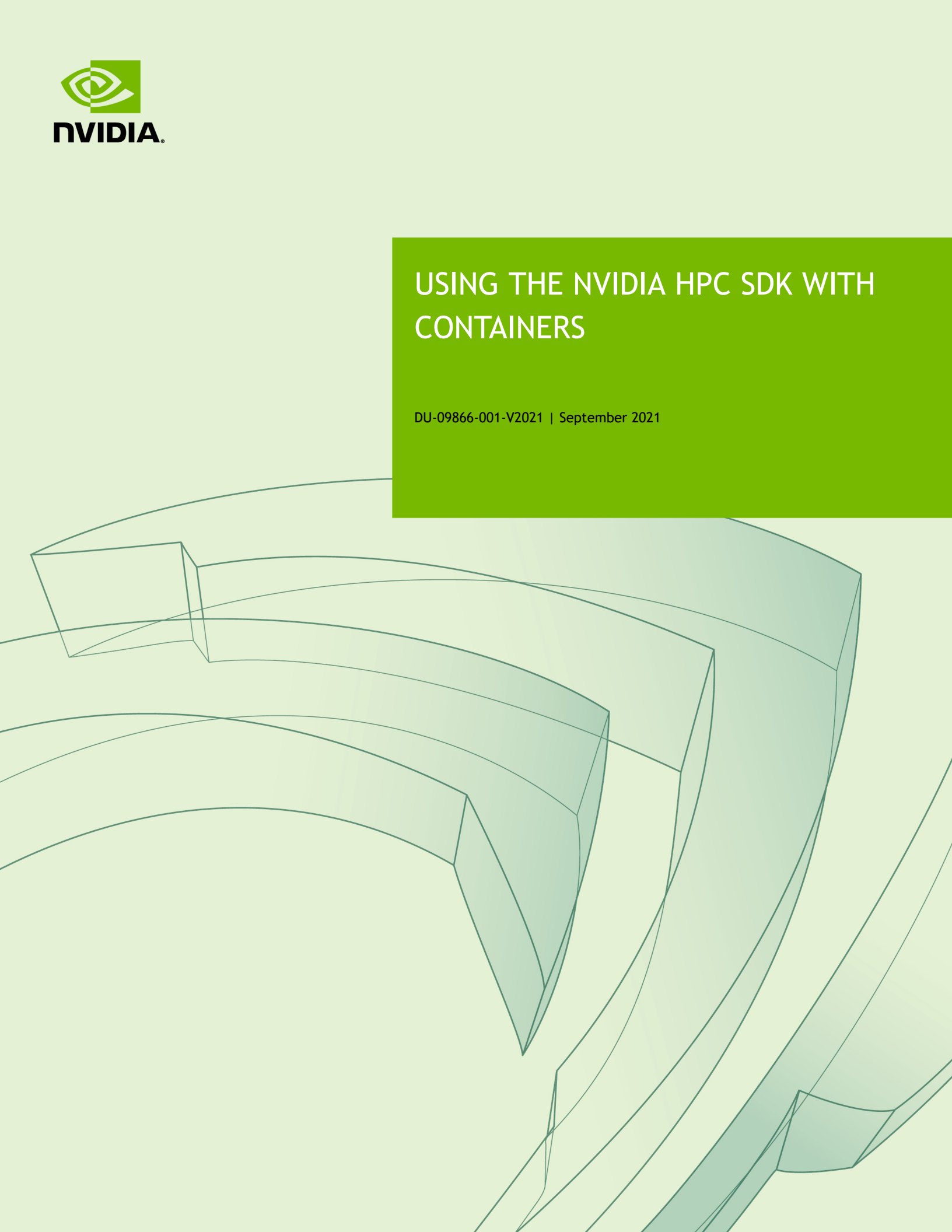

USING THE NVIDIA HPC SDK WITH CONTAINERS

DU-09866-001-V2021 | September 2021

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.

- Docker, described further in the section Docker Containers, helped popularize Linux containers and is used widely.
- Singularity, described further in the section Singularity Containers, addresses some of the challenges with using containers in HPC environments and is available at most HPC centers.

Chapter 2.
DEVELOPING WITH CONTAINERS

All the examples in this section are shown for both Docker and Singularity, and are also generally applicable to other container runtimes. The CloverLeaf mini application is used as a demonstration workflow.

Using containers for development is an alternative to the traditional host-based development workflow. With a container, the development environment can be precisely defined by the user. For instance, the container can use a different libc than the host or include additional libraries not available on the host.

The examples use the HPC SDK container images available from NGC. These freely available images are the best way to get started using the HPC SDK and containers.

2.1. Docker Containers

To start an interactive Docker development environment run:

```
$ sudo docker run --rm -it --runtime=nvidia --user $(id -u):$(id -g) --volume
 $(pwd):/source --workdir /source nvcr.io/nvidia/nvhpc:21.9-devel-centos7
```

Since containers are ephemeral, this command mounts the source code directory from the host as **`/source`** inside the container (**`--volume`**) and defaults to this directory when the container starts (**`--workdir`**). This assumes the command to start the container is run from the location where the CloverLeaf source code was checked out. Any changes to the source code or builds 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 object files and other files created inside the container in the **`/source`** directory would be owned by **`root`**.

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

2.2. Singularity Containers

To start an interactive Singularity development environment:

```
$ singularity shell --nv docker://nvcr.io/nvidia/nvhpc:21.9-devel-centos7
```

Starting a Singularity container does not require superuser privileges, unlike Docker. Also, 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. This assumes the command to start the container is run from the location where the CloverLeaf source code was checked out. If the CloverLeaf source is located somewhere else, start the container using

```
$ singularity shell --nv -B <path-to-source>/CloverLeaf-OpenACC:/source --pwd /
source docker://nvcr.io/nvidia/nvhpc:21.9-devel-centos7
```

2.3. Example Using CloverLeaf

From the shell inside the Docker or Singularity container, run **`make COMPILER=PGI`** to build CloverLeaf, and then run it with the default small dataset **`(mpirun -n 1 clover_leaf)`**.

By default, CloverLeaf is configured to build for Pascal (i.e., compute capability 6.0) devices. Running CloverLeaf on non-Pascal GPUs will generate an error similar to the following. Refer to the section Multi-Architecture Support for more information.

```
This file was compiled: -ta=tesla:cc60
Rebuild the application with -ta=tesla:cc70 to use NVIDIA Tesla GPU 0
```

The clover_leaf binary will most likely not run on the host outside the container due to library dependencies that are only satisfied inside the container. To distribute the resulting binary outside a container libraries could be linked statically, or the necessary libraries could be redistributed with the binary. However, please see the section on building containerized applications for a better solution.

Chapter 3.
PROFILING WITH CONTAINERS

In general, profiling a workload running in a container is not significantly different than profiling a workload on bare metal. The Nsight Compute and Nsight Systems profilers are part of the HPC SDK.

A typical workflow is to iterate profiling and code modifications until the desired performance is achieved. Nsight Systems provides low overhead, comprehensive workload-level performance analysis and is the place to start profiling. Once the macro level performance is understood, detailed CUDA kernel performance is available from Nsight Compute.

To start an interactive Docker profiling environment run using CloverLeaf source located on the host:

```
$ sudo docker run --rm --runtime=nvidia --cap-add=SYS_ADMIN --volume $(pwd):/
source --workdir /source nvcr.io/nvidia/nvhpc:21.9-devel-centos7
```

Docker containers need to be started with the command line option **`--cap-add=SYS_ADMIN`** to enable capabilities required for profiling. No additional options are needed when profiling workloads running in Singularity containers.

To start an interactive Singularity profiling environment run:

```
$ singularity shell --nv docker://nvcr.io/nvidia/nvhpc:21.9-devel-centos7
```

From the shell inside the Docker or Singularity container, run **`make COMPILER=PGI`** to build CloverLeaf, if it was not built in the previous section.

Also from the shell inside the container, the Nsight Systems command line interface can be used to profile CloverLeaf with the default dataset:

```
$ mpirun -n 1 nsys profile --stats=true -t openacc,cuda,nvtx,osrt ./clover_leaf
```

Nsight Systems will output some high level statistics such as the top CUDA kernels, in addition to a **`qdrep`** report that can be visualized using the Nsight Systems graphical user interface.

Similarly, the Nsight Compute command line interface can be used to profile CloverLeaf from a shell inside the Docker or Singularity container. Nsight Systems revealed that the **`update_halo_kernel`** kernel is one the top CloverLeaf kernels. To profile the first 20 instances of the kernel run:

```
$ mpirun -n 1 nv-nsight-cu-cli -k update_halo_kernel -c 20 ./clover_leaf
```

For more information, please consult the Nsight Systems or Nsight Compute documentation.

Chapter 4. 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.

For CloverLeaf, a multi-stage build can reduce the Singularity container image size by over 32X, from 4.7 GB to 146 MB.

4.1. Building Containerized Applications with Docker

To build a container image with Docker:

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

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

The following Dockerfile builds a container from the local CloverLeaf source checkout:

```
# Build stage
FROM nvcr.io/nvidia/nvhpc:21.9-devel-centos7 AS build

# build CloverLeaf
COPY CloverLeaf-OpenACC /source
RUN cd /source && make COMPILER=PGI

# Runtime stage
FROM nvcr.io/nvidia/nvhpc:21.9-runtime-cuda11.0-centos7

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

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

The build stage of the Dockerfile is based on the HPC SDK development image from NGC. The local CloverLeaf source code is copied into the container at **`/source`**, and then built. The runtime stage is based on the smaller HPC SDK runtime image, also from NGC. Three HPC SDK runtime images are available, for CUDA 10.1, 10.2, and 11.0; select the version corresponding to the CUDA version used to build CloverLeaf, assumed here to be CUDA 11.0. The **`clover_leaf`** binary and the sample input datasets are copied from the build stage.

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.

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

To generate the container image from this Dockerfile:

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

Alternatively, the CloverLeaf source could be checked out from the GitHub repository:

```
# Build stage
FROM nvcr.io/nvidia/nvhpc:21.9-devel-centos7 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

# Runtime stage
FROM nvcr.io/nvidia/nvhpc:21.9-runtime-cuda11.0-centos7

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
```

After checking out the git repository, the rest of the Dockerfile is virtually identical to the one using the local CloverLeaf checkout.

To generate the container image from this Dockerfile:

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

In either the local source directory or git repository cases, the containerized CloverLeaf can be run using the following command, where <tag> is either **`local`** or **`git`**:

```
$ sudo docker run --rm --runtime=nvidia cloverleaf:<tag> mpirun -n 1 --allow-
run-as-root clover_leaf
```

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`**.

```
$ sudo docker run --rm --runtime=nvidia --volume $(pwd)/clover_bm32_short.in:/
clover.in cloverleaf:<tag>  mpirun -n 1 --allow-run-as-root clover_leaf
```

4.2. 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 following Singularity.def builds a container from the local CloverLeaf source checkout:

```
# Build stage
BootStrap: docker
From: nvcr.io/nvidia/nvhpc:21.9-devel-centos7
Stage: build

%files
  CloverLeaf-OpenACC /source

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

  # build CloverLeaf
  cd /source
  make COMPILER=PGI

# Runtime stage
BootStrap: docker
From: nvcr.io/nvidia/nvhpc:21.9-runtime-cuda11.0-centos7

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

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

The instructions 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.

To generate the container image from this Singularity definition file:

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

Alternatively, the CloverLeaf source could be checked out from the GitHub repository:

```
# Build stage
BootStrap: docker
From: nvcr.io/nvidia/nvhpc:21.9-devel-centos7
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

# Runtime stage
BootStrap: docker
From: nvcr.io/nvidia/nvhpc:21.9-runtime-cuda11.0-centos7

%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
```

In either the local source directory or git repository cases, the containerized CloverLeaf can be run using the following command, where <label> is either local or git:

```
$ singularity run --nv cloverleaf-<label>.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
```

4.3. 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:21.9-devel-centos7',
 _as='build')

# build CloverLeaf
Stage0 += generic_build(build=['make COMPILER=PGI'],
    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:21.9-runtime-cuda11.0-centos7')

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 customize the set of HPC SDK packages to install in the container or use a professional edition in place of the community edition that will be available via NGC.

For example, the following recipe first installs the HPC SDK into a container, then builds CloverLeaf, and finally automatically generates the container instructions to copy the HPC SDK runtime into the runtime container stage. The size of the final container image is considerably smaller than the previous examples using the NGC runtime images because a smaller subset of the runtime is included in the final image.

```
# Build stage
Stage0 += baseimage(image='centos:7', _as='build')

# NVIDIA HPC SDK
Stage0 += nvhpc(eula=True, redist=['compilers/lib/*'])

# Additional development tools
Stage0 += packages(ospackages=['git', 'make'])

# CloverLeaf
Stage0 += generic_build(build=['make COMPILER=PGI'],
    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='centos:7')
Stage1 += Stage0.runtime()
Stage1 += environment(variables={'PATH': '/opt/CloverLeaf-OpenACC/bin:$PATH'})

# nvidia-container-runtime
Stage1 += environment(variables={
  'NVIDIA_VISIBLE_DEVICES': 'all',
  'NVIDIA_DRIVER_CAPABILITIES': 'compute,utility',
  'NVIDIA_REQUIRE_CUDA': '"cuda>=10.1 brand=tesla,driver>=384,driver<385
 brand=tesla,driver>=396,driver<397 brand=tesla,driver>=410,driver<411"'})
```

The HPCCM GitHub repository includes sample recipes for real world applications such as **`GROMACS`**, **`LAMMPS`**, and **`MILC`**.

Chapter 5. BEST PRACTICES

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

5.1. 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.

5.2. 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
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.

For instance, HPCCM generates a single layer to download, build, install, and cleanup CloverLeaf:

```
RUN mkdir -p /var/tmp && cd /var/tmp && git clone --depth=1 https://github.com/
UoB-HPC/CloverLeaf-OpenACC.git CloverLeaf-OpenACC && cd - && \
    cd /var/tmp/CloverLeaf-OpenACC && \
    make COMPILER=PGI && \
    mkdir -p /opt/CloverLeaf-OpenACC && \
    cd /var/tmp/CloverLeaf-OpenACC && \
    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 && \
    rm -rf /var/tmp/CloverLeaf-OpenACC
```

5.3. 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.

To support multiple GPU compute capabilities, replace **`cc60`** in the CloverLeaf Makefile with **`ccall`**. The default values in the CloverLeaf Makefile can be overridden on the command line, as shown in this HPCCM recipe snippet.

```
Stage0 += generic_build(build=['make COMPILER=PGI FLAGS_PGI="-Mpreprocess -fast
 -acc -Minfo=acc -ta=tesla,ccall"'],
    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')
```

The resulting container will run optimally on any GPU.

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']:
  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 Arm, Power, and X86 processors. The base image is processor architecture specific, but HPCCM handles the other differences automatically. By using conditionals in the recipe, the few differences can be handled in a single recipe. For example, a single recipe capable of generating CloverLeaf containers for X86 (default) and Power (if the HPCCM command line argument **--userarg power=True** is specified):

```
# Build stage
if USERARG.get('power', None):
  image='ppc64le/centos:7'
else:
  image='centos:7'

Stage0 += baseimage(image=image, _as='build')

# NVIDIA HPC SDK
Stage0 += nvhpc(eula=True, redist=['compilers/lib/*'])

# Additional development packages
Stage0 += packages(ospackages=['git', 'make'])

# build CloverLeaf
Stage0 += generic_build(build=['make COMPILER=PGI'],
    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=image)
Stage1 += Stage0.runtime()
Stage1 += environment(variables={'PATH': '/opt/CloverLeaf-OpenACC/bin:$PATH'})

# nvidia-container-runtime
Stage1 += environment(variables={
  'NVIDIA_VISIBLE_DEVICES': 'all',
  'NVIDIA_DRIVER_CAPABILITIES': 'compute,utility',
```

```
  'NVIDIA_REQUIRE_CUDA': '"cuda>=10.1 brand=tesla,driver>=384,driver<385
 brand=tesla,driver>=396,driver<397 brand=tesla,driver>=410,driver<411"'})
```

5.4. 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'],
        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