Shifter: Fast and consistent HPC workflows using containers

Lucas Benedicic*, Felipe A. Cruz, Thomas C. Schulthess
Swiss National Supercomputing Centre, CSCS
Lugano, Switzerland
*Email: *benedicic@cscs.ch*

***Abstract*—In this work we describe the experiences of building and deploying containers using Docker and Shifter, respectively. We present basic benchmarking tests that show the performance portability of certain workflows as well as performance results from the deployment of widely used non-trivial scientific applications. Furthermore, we discuss the resulting workflows through use cases that cover the container creation on a laptop and their deployment at scale, taking advantage of specialized hardware: Cray Aries interconnect and NVIDIA Tesla P100 GPU accelerators.**

***Keywords*-HPC systems, GPU, GPGPU, containers, Docker, Shifter.**

I. Introduction

Containers are packaged applications in the form of a standardized unit of software that is able to run on multiple platforms. In a nutshell, a container packs a software application with its filesystem containing the whole environment that is needed for its execution, i.e., code, runtime tools, and software dependencies. At run time, a container will share the operating system kernel of the host machine allowing containers to start instantly and have a smaller footprint than other virtualization technologies like hypervisors [1].

Containers have already had a positive impact on developers and operations alike as the technology:

- simplifies the work of software developers by streamlining application packaging as a portable unit, making building and testing software easier and faster;
- provides self-contained and isolated applications with a small footprint and low runtime overheads that results in software that is easier to distribute and deploy.

The use of containers in High Performance Computing (HPC) has so far and for the most part been exploratory. High performance software is traditionally built directly on the target system in order to take advantage of the specialized hardware. This and other needs that are particular to HPC have delayed the adoption of container technologies within such environments. However, efforts exist to develop container runtimes that specifically target the needs of HPC. One such project is Shifter [2], a container runtime tailored for HPC that has been extended to give executing containers access to host-specific libraries and tools, also enabling hardware-accelerated features. The work in [3] shows that a Docker/Shifter workflow can provide performance portability, natively supporting Graphic Processing Unit (GPU) accelerators and fast network interconnects, from a workstation to an HPC system like Piz Daint. The possibility of consistently delivering such workflows could truly transform the building, testing, distribution, and deployment of scientific software, enabling qualitatively better computing workflows.

Leveraging further into their possibilities, containers could also be used to provide a complete software stack to solve a particular problem. Using such specialized containers enables the delivery of readily-available environments that provide an HPC-compatible software stack. The users would quickly extend such containers to match a particular problem instance directly on their workstations. Such specialized containers can be valuable to traditional HPC users, but should be of particular value to other scientific domains, e.g. data sciences communities.

The remainder of this paper is organized as follows. Section II gives an overview of the Docker and Shifter technologies. Section III presents a basic workflow for building Docker containers and deploying them with Shifter. Section IV presents a selection of use cases that involve containers highlighting different user workflows.

II. Background

In this section we provide a brief overview of a workflow that consists of: (1) building and testing containers using a standard laptop, and then (2) deploying and executing them on an HPC infrastructure while achieving high-performance. Since this Section is not meant as an in-depth description of the technologies that enable these workflows, we refer the reader to Docker [4] and Shifter [2][3] for a comprehensive discussion on these topics.

A. Docker

Containers are a type of virtualization that operates at Operating System (OS) level, abstracting the containerized application from the hardware over which it is run. To achieve this, container virtualization interfaces containers with the host system through OS kernel system calls. The straightforward benefit of virtualizing at the OS level is that containerized applications have a low processing overhead and can run on most Linux-based platforms.

Container virtualization works by packaging an application into an image that bundles a software application along

with its complete environment, i.e., with everything needed by the application to ensure its correct execution. The de-facto format is the Docker image format [4], which is part of the Docker open-source project that also provides the tools required to build and pack applications into the self-contained environment of a container image.

Besides the fact that containers are lightweight and flexible, there are many other benefits that containers bring to software development and deployment:

- Simple and quick iterations. The lightweight and flexibility of containers along side Docker tools makes developing, building, and deploying complex software environment configurations simple.
- Docker containers are portable units of software. Software containers are both hardware- and platform-agnostic making them portable, standardized units of software.
- Portability and rapid deployment. Container images can be used across platforms. Moreover, containers can be started as quickly as a native process running on the target system.
- Shareable and reproducible self-contained units of software. Containers are ready to run units of software and as such they can be easily shared, and their results reproduced.
- Increased productivity. The low overhead of containers allows for interactive development using the same environment as the production system.

The core of the Docker workflow revolves around two concepts: an *image*, which can be understood as an immutable snapshot of the application and its software environment; and the *container*, a runtime instance of an image.

The workflow for working with Docker images and containers using the `docker` application centers around four core operations [5]:

Build Makes use of recipe files called *dockerfiles* that describe all the steps needed to build and run an application. The `docker build` tool allows for version-control driven image creation, and custom images that contain the application and all its dependencies can be easily created using *dockerfiles*. Command: `docker build -t <image>:<tag>`

Push Enables the storing of images into both private and community-driven registries. A registry is a service which hosts and distributes images. The default registry server for Docker is Docker Hub [6] which is hosted by Docker Inc. Command: `docker push <image>:<tag>`

Pull A given container image can be retrieved from the registry so it can be modified, build upon, or run locally. Command: `docker pull <image>:<tag>`

Run Images that are stored locally can be used to create a container instance by running them. Command: `docker run <image>:<tag> <path to containerized app>`

B. Shifter

Shifter is a lightweight container runtime that provides to HPC systems with an efficient and secure approach for end-users to execute Docker images. Moreover, recent work on Shifter [3] further extended its functionality to provide containerized applications with a transparent and easy way to access GPU accelerators and fast network interconnects. The Shifter runtime provides its functionality by performing the following operations when launching a containerized application:

- instantiates the container using two sources: the software environment from the image; and, the host-specific resources that have been defined through the Shifter's configuration file `udiroot.conf`;
- provides containers with transparent access to specialized hardware by correctly bind mounting host-specific libraries (CUDA and Message Passing Interface (MPI));
- launches the containerized application in user-space, specifically using the credentials and privileges of the user;
- integrates with the resource manager (e.g., SLURM, ALPS) to efficiently run HPC applications at scale.

Therefore, HPC users of Docker containers can take advantage of all the benefits of a workflow that uses Docker to build and test container images, and leverages on Shifter to deploy containers on HPC systems while attaining high-performance.

III. User workflow

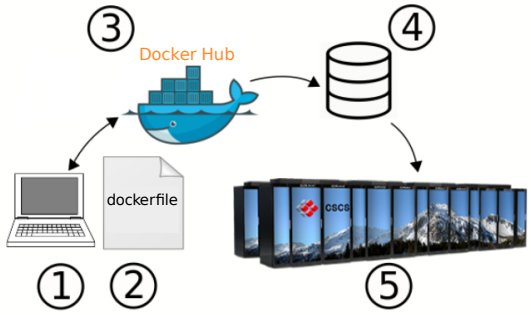

Figure 1. User workflow for building Docker containers and deploying them with Shifter on HPC systems. Summary of the workflow steps: 1) build container image; 2) test container with Docker; 3) Push container to Docker Hub; 4) Pull image into HPC system; 5) Deploy container using Shifter.

The standard workflow for using Docker and Shifter is depicted in Figure 1 and its five steps can be summarized as follows:

1) Build the container image using the tools provided by the Docker. This step takes place on the researcher's workstation using *dockerfile* (the container image descriptor) and the `docker build` tool. Moreover, the Docker image can be based on the Linux distribution of choice, enabling the tools that are most familiar to the user.
2) The researcher tests the image to verify the correctness of the software stack and the scientific simulation by running it locally using the `docker run` command. The use of *dockerfiles*, `docker build` and `docker run` allow users to rapidly iterate on the *dockerfile* until the application and its environment is correctly configured.
3) Once an image has been validated using Docker, the user can upload it to a remote registry using `docker push`. Although the default registry service used by Docker is the Docker Hub, a service provided by Docker Inc. The use of third-party repositories is also supported.
4) Images from the registry can be pulled by an HPC system using `shifterimg` tool. When an *image pull* takes place, the image is downloaded from the registry into the Image Gateway service, where it is flattened, converted into squashFS format (compressed read-only file system for Linux), and stored in the parallel file system.
5) Using the Shifter Runtime, the container can then be instantiated using data from both the local container image and host-specific resources as specified by Shifter's configuration file.

The steps described above present a workflow that merges some of the best features of `docker` with `shifter`. On the one hand, steps one to three of the above workflow show that Docker can be effectively used to build and test the container solely using the researcher laptop. Moreover, these tools enable the use and deployment of software environments that are familiar to the end-user. On the other hand, steps four and five leverage on the image that was created on the workstation, moving it to the HPC system using `shifterimg pull` and then deploying it using the `shifter` command. Moreover, the use of the Shifter enables the container to take advantage of the specialized hardware available on the HPC site.

A. Example

To illustrate the workflow described above we first present a simple example. The objective of this example is to build a potentially complex software stack on a container image that can make use of the CUDA toolkit to access the GPU resource available on the host system. To this end we containerize the `deviceQuery` application, which is a CUDA application performs the properties listing of the CUDA devices present on the host system. This test effectively shows that a hardware- and software-agnostic containers can access host-specific GPU resources through Shifter.

We start by preparing the *dockerfile* document that contains the commands needed to assemble the image. The example *dockerfile* is presented on Listing 1.

Listing 1. Example CUDA dockerfile

```
1 FROM nvidia/cuda:8.0
2
3 RUN apt-get update && apt-get install -y  \
4         --no-install-recommends           \
5         cuda-samples-$CUDA_PKG_VERSION && \
6     rm -rf /var/lib/apt/lists/*
7
8 RUN (cd /usr/local/cuda/samples/1_Utilities/
     deviceQuery && make)
```

Of *line 1* of the Listing 1, the `FROM` command sets the base image to the one provided by NVIDIA (`nvidia/cuda:8.0`). Upon this base image all subsequent instructions will be applied on. By looking at the *dockerfile* of image `nvidia/cuda:8.0` available from [7] we know that this image is based on Ubuntu 16.04 Linux distribution to which the runtime and utilities of CUDA have been installed. Using this image as our starting point greatly simplifies the software environment setup of our own image. However, the container image prepared by NVIDIA does not provide the deviceQuery and other utilities. To enable these applications we can obtain the application source code using apt-get (see *lines 3 to 6* of Listing 1). Finally, *line 8* performs the compilation of our test application.

The next step is to build the image and test the container on a laptop using nvidia-docker [8], an extension to the Docker runtime developed by NVIDIA to provide access to the GPU. These steps are shown in Listing 2.

Listing 2. Build dockerfile and test container on laptop

```
1 $ nvidia-docker build -t "ethcscs/dockerfiles
     :cudasamples8.0" .
2
3 $ nvidia-docker run ethcscs/dockerfiles:
     cudasamples8.0 /usr/local/cuda/samples/1
     _Utilities/deviceQuery/deviceQuery
4
5 $ nvidia-docker push ethcscs/dockerfiles:
     cudasamples8.0
```

From Listing 2 we observe the following: *line 1* shows the command used for building the image, the `-t` flag sets the name and tag of the image, while the dot (`.`) indicates to use the *dockerfile* from the current directory. After that we can run the containerized application using the `run` command as it is shown in *line 3*; on *line 5* we push the image to Docker Hub.

The output of the containerized `deviceQuery` application running from the container using `nvidia-docker` on a Laptop is shown in Listing 3. The output has been trimmed to highlight the GPU devices detected.

Listing 3. Output of `deviceQuery` on Laptop

```
1 ./deviceQuery Starting...
2 [...]
3 Detected 1 CUDA Capable device(s)
4
5 Device 0: "Quadro K1100M"
6 [...]
```

We now briefly compare the pulling and execution of the same container image on an HPC system. Consider Listing 4, where we pull and execute using the tools provided by Shifter on an HPC system: *line 1* shows the usage of `shifterimg` to download the image; *line 3* makes use of SLURM's `srun` command to run the containerized `deviceQuery` application using `shifter`.

Listing 4. `deviceQuery` image on HPC system

```
1 $ shifterimg pull ethcscs/dockerfiles:
     cudasamples8.0
2
3 $ srun -C gpu shifter --image=ethcscs/
     dockerfiles:cudasamples8.0 ./deviceQuery
```

Listing 5 shows the output after executing the container with Shifter on Piz Daint, and the containerized application correctly detects the NVIDIA Tesla P100 GPU available on the system.

Listing 5. Output of deviceQuery on HPC system

```
1 ./deviceQuery Starting...
2 [...]
3 Detected 1 CUDA Capable device(s)
4
5 Device 0: "Tesla P100-PCIE-16GB"
6 [...]
```

IV. HPC WORKFLOW WITH CONTAINERS

Achieving high-performance and portability of containerized applications by running Docker containers with Shifter has been demonstrated in [3]. Hence instead of focusing the discussion on these aspects we will center on the different HPC workflows that are enabled by the use of Docker and Shifter. This section highlights two crucial aspects of the workflows: first, the creation of container images; second, using Docker containers to access HPC resources. Moreover, through this section we will cover the following:

- Building images with complex dependencies.
- Using container images that have already been created by a third party.
- Launching containerized applications with SLURM.
- Using MPI and fast network interconnects.
- Using CUDA to access GPUs.
- Applications that use both CUDA and MPI.
- The usage of containers as portable compilation units.

The rest of this section is organized as follows: first we present the methodology for building and running containers; then we present a variety of use cases that highlight the points above using different workflows.

A. Methodology

We evaluate a number of containerized applications from images that have been built with Docker on a Laptop or that are provided by trusted third parties when possible. Shifter is only used to run the containerized applications on HPC infrastructure. We make use two systems throughout this section: a Laptop to build and test Docker images, and Piz Daint to run the containerized applications. The configuration of these systems is as follows:

Laptop specification

- Model: Lenovo® W540 mobile workstation.
- Processor: Intel®Core™i7-4700MQ processor.
- Memory: 8 GB of RAM.
- GPU: Nvidia®Quadro™K110M GPU with 2 GB of memory.
- OS: CentOS 7 with Linux kernel 3.10.0.
- Libraries: CUDA 8.0, MPICH 3.2.

Piz Daint specification

- Model: hybrid Cray XC50/XC40.
- Processor: Intel® Xeon® E5-2690v3 processor.
- Memory: 64 GB of RAM.
- GPU: Nvidia® Tesla® P100 with 16 GB of memory.
- Network: Cray Aries interconnect under a Dragonfly topology.
- OS: Cray Linux Environment 6.0 UP02 [9] with Linux kernel 3.12.60.
- Libraries: CUDA 8.0, and Cray's MPI Library MPT 7.5.0.

Piz Daint is currently the eighth fastest supercomputer in the world [10] and provides its users with support to execute Docker containers through Shifter.

B. N-body: CUDA application

A fast N-body simulation fully implemented in CUDA was developed by NVIDIA as part of the CUDA Software Development Kit [11]. This application simulates the dynamical evolution of a system of particles under the influence of gravity. It is compute intensive and makes efficient use of GPUs to perform the calculations in parallel.

Listing 6. N-body `dockerfile`

```
1 FROM nvidia/cuda:8.0
2
3 RUN apt-get update && apt-get install -y --no
     -install-recommends \
4         cuda-samples-$CUDA_PKG_VERSION && \
5     rm -rf /var/lib/apt/lists/*
6
7 RUN (cd /usr/local/cuda/samples/5_Simulations
     /nbody && make)
```

The preparation of the container image using a `dockerfile` is almost identical to the one presented in Listing 2. However, in Listing 6 we compile the N-body application instead (see *line 7*).

We can now build and push the image into the main registry, as seen in *lines 1* and *3* of Listing 7.

Listing 7. Build image and push it to the main Docker registry

```
1 $ docker build -t "ethcscs/dockerfiles:
      cudasamples8.0" .
2
3 $ docker push ethcscs/dockerfiles:
      cudasamples8.0
```

Once the image has been built from the `dockerfile` and pushed to the registry it is possible to download the container image to Piz Daint using the `shifterimg` tool. The image can be deployed using the `shifter` tool, as it can be seen in Listing 8 *line 3* and *line 5* respectively. Note that in order to use Shifter in Piz Daint it is first necessary to load the module that makes the tool available, as it can be seen in *line 1* of Listing 8.

Listing 8. Pull and run N-body container

```
1 $ module load shifter/17.03.00
2
3 $ shifterimg pull ethcscs/dockerfiles:
      cudasamples8.0
4
5 $ srun -N 1 -C gpu shifter --image=ethcscs/
      dockerfiles:cudasamples8.0 /usr/local/
      cuda/samples/bin/x86_64/linux/release/
      nbody -benchmark -fp64 -numbodies=200000
```

C. TensorFlow: a third-party image using CUDA

TensorFlow is an Open Source software framework for machine learning that was developed by Google. Computations in TensorFlow are expressed through its API as data flow graphs and it provides an implementation to run these computations on a range of systems. Moreover it is able to make use of GPUs using CUDA. For convenience, the developers of TensorFlow also distribute the TensorFlow application and all dependencies with support for NVIDIA CUDA as a Docker container image through Docker Hub.

We use the official TensorFlow Docker image that is available through Docker Hub under the tag 1.0.0-devel-gpu-py3 (see Listing 9). The official image is based on the Ubuntu 14.04 Linux distribution with the following packages installed: Python 3.4.3, Nvidia CUDA Toolkit 8.0.44, and NVIDIA cuDNN 5.1.5.

Listing 9. Download official TensorFlow image

```
shifterimg pull docker:tensorflow/tensorflow
    :1.0.0-devel-gpu-py3
```

We use the MNIST database [12], a collection of images that represent handwritten digits, to test the training an image recognition model. In Listing 10 we obtain the model from the TensorFlow repository (*line 1*) and data (*line 3*), storing the model and data on the home folder on Piz Daint. The implemented test training model makes use of 60,000 training image examples and 10,000 test examples.

Listing 10. TensorFlow model and data for MNIST

```
1 wget https://raw.githubusercontent.com/
     tensorflow/models/master/tutorials/image/
     mnist/convolutional.py
2
3 cp -r /apps/daint/UES/mnist/data/ .
```

We can now run the container as seen in Listing 11 on a single node of Piz Daint: we schedule the container to run with SLURM using the Shifter runtime.

Listing 11. Execution of containerized TensorFlow

```
srun -C gpu -N1 -n1 shifter --image=
    tensorflow/tensorflow:1.0.0-devel-gpu-py3
     python3 convolutional.py
```

Executing the container through Shifter will make the GPU NVIDIA P100 available to the container. Table I compares the wall-clock time of running the MNIST TensorFlow test on the Laptop vs Piz Daint. For the Laptop we present the time in seconds while for Piz Daint we present the relative speedup with respect to the Laptop results.

Table I
CONTAINERIZED TENSOFFLOW PERFORMANCE RESULTS ON THE LAPTOP PRESENTED IN SECONDS AND THE RELATIVE SPEEDUP OBTAINED ON A SINGLE NODE OF PIZ DAINT FOR THE MNIST TEST.

	Laptop	Piz Daint
MNIST	613	17.17x

D. Trilinos: library using MPI

Trilinos [13] is an open-source project that provides a software framework for solving scientific and engineering application by using specialized software packages. Trilinos packages are self-contained and each has their own set of dependencies. The Epetra package implements parallel linear algebra solvers and provides a core foundation for Trilinos-based applications.

In this case we build a container with Trilinos' Epetra package and use one of its performance tests to demonstrate a non-trivial MPI workflow. `BasicPerfTest` from Epetra builds a system of equations for a 2D PDE finite difference problem and solves it using a LU-based solver.

The *dockerfile* used to prepare the Trilinos container image can be seen in Listing 16. Please note that due to the size of the *dockerfile* this has been added as an appendix to this paper. As it can be seen in the Trilinos *dockerfile*, our image is built based on Debian Jessie Linux distribution onto which we install basic development tools, MPICH 3.1.5, the Trilinos Epetra package, and finaly the `basicPerfTest` benchmark. As we have seen before, building, running, and pushing the container image starting from the *dockerfile* is straightforward by using the commands: `docker build`, `docker run`, `docker push` (see Listing 12).

Listing 12. Using `docker` to build from *dockerfile*, run, and push container image to Docker Hub.

```
1 $ docker build -t ethcscs/dockerfiles:
      trilinos-epetrampi-benchmark .
2
3 $ docker run --rm ethcscs/dockerfiles:
      trilinos-epetrampi-benchmark
4
5 $ docker push ethcscs/dockerfiles:trilinos-
      epetrampi-benchmark
```

Pulling and running the container image is as what we saw with the previous examples with the only difference that this example does not make use of the GPU but the fast network interconnect through MPI. Listing 13 shows the following: *line 1* downloads the container from Docker Hub; *line 3* uses SLURM to allocate 128 MPI processes on four nodes of the multicore partition on Piz Daint (in this partition, each node has 36 cores and no GPU). Also, on the SLURM allocation we make use of two Shifter-specific options that set the container image to be used (`--image`) and enable Shifter's MPI-support (`--shifter-mpi`); *line 5* uses SLURM `srun` to schedule the execution of the containerized application with Shifter. Please note that the image and MPI support were set with `salloc`.

Shifter provides the container with access to the compute node's MPI implementation in order to use the Aries Interconnect of Piz Daint. To take advantage of this feature, the MPI installed in the container (and dynamically linked to your application) has to be ABI-compatible with the compute node's MPI on Piz Daint. To meet the required ABI-compatibility, we recommend that the container application uses one of the following MPI implementations:

- MPICH v3.1 (Feburary 2014)
- MVAPICH2 2.2 (September 2016)
- Intel MPI Library v5.0 (June 2014)

Listing 13. Pull and run the Trilinos container image with Shifter

```
1 $ shifterimg pull ethcscs/dockerfiles:
      trilinos-epetrampi-benchmark
2
3 $ salloc -C mc -n 128 -N 4 --image=ethcscs/
      dockerfiles:trilinos-epetrampi-benchmark
      --shifter-mpi
4
5 $ srun shifter /opt/Trilinos-trilinos-release
      -12-10-1/build/packages/epetra/test/
      BasicPerfTest/Epetra_BasicPerfTest_test.
      exe 225 450 16 8 5 -v
```

Results of running the `BasicPerfTest` on both the Laptop system (4 cores) and Piz Daint (4 nodes on 32 cores each) are presented in Table II.

E. PyFR: Complex dependency set with CUDA and MPI

PyFR [14] is a Python code for solving high-order computational fluid dynamics on unstructured grids. The numerical method used by this software has shown efficient execution on GPUs and parallel scalability on HPC systems. Moreover, this project was one of the Gordon Bell Prize finalist in 2016.

Table II
CONTAINERIZED TRILINOS PERFORMANCE RESULTS ON THE LAPTOP (USING 4 CORES) PRESENTED MFLOPS AND THE RELATIVE SPEEDUP OBTAINED PIZ DAINT.

	Norm2	Dot	Updates
Laptop	3176.41	3306.74	1843.24
Piz Daint	30.9x	18.4x	15.9x

We base the Docker image for this application on the official Ubuntu 16.04 Docker image to which we installed basic software development tools and all PyFR dependencies: Python 3.5.2, METIS, NVIDIA CUDA Toolkit 7.5, and MPICH 3.1.4. The `dockerfile` can be seen in Listing 17, which due to its length has been added as part of the appendix. Building, running, and pushing the Docker container image is as we have seen in previous examples of these steps. In the following, we will discuss the execution of the PyFR application on multiple nodes with both GPU and MPI support enabled.

Listing 14. Download PyFR container image with Shifter

```
$ shifterimg pull ethcscs/dockerfiles:pyfr
    -1.5.0-cuda7.5
```

As an illustration, we will run the containerized application from the PyFR image that we just downloaded. For this, we will schedule a SLURM job that makes use of two GPU nodes of Piz Daint (see *line 1* of Listing 15). As with previous examples, SLURM options are set to enable Shifter's use of GPU, MPI, and the image. As it can be seen in *line 3*, to run the MPI job SLURM `srun` is set to the number of processes (`-n 2`) and uses `shifter` to execute the `pyfr` application along with its command-line options.

Listing 15. Run PyFR container image with Shifter

```
1 $ salloc -N 2 -C gpu --image=ethcscs/
      dockerfiles:pyfr-1.5.0-cuda7.5 --shifter-
      mpi
2
3 $ srun -n 2 shifter pyfr run -b cuda -p
      euler_vortex_2d.pyfrm euler_vortex_2d.ini
```

The chosen test case calculates the 3D flow over a turbine blade. The simulation domain was discretized with a mesh composed of $114,265$ cells and $1,154,120$ points using a total of 10GB of memory. The simulation itself makes use of the GPU through a CUDA backend and can run in parallel on multiple nodes using MPI. Due to the size of the problem the Laptop system was not able to run this simulation. However, the Laptop system was indeed used to build the container and to test the application through some trial simulation test. On Piz Daint, the simulation was carried out using multiple nodes and its parallel efficiency is reported in Table III.

F. Portable compilation units

The performance of some applications can strongly depend on the targeted optimization of the underlying libraries.

Table III
PyFR simulation of a 3D flow solver over a turbine blade with a mesh discretization of 1, 154, 120 mesh points, problem size suitable for up to 4 nodes. Results present the parallel efficiency of strong scaling the test case up to 16 nodes on Piz Daint.

System	1 GPU	2 GPUs	4 GPUs	8 GPUs	16 GPUs
Piz Daint	1.0	0.975	0.964	0.927	0.874

A good example of such situation is the Linpack benchmark [15], which measures a system's floating point performance by solving a dense n by b system of linear equations. HPL is a portable C implementation of HPLinpack, and requires Basic Linear Algebra Subprograms (BLAS) and MPI libraries. Moreover, the most important BLAS subroutine from performance considerations is Double-precision General Matrix Multiply (DGEMM). Therefore, it is crucial for Linpack to have access to an optimized BLAS library. However, there is no ABI compatibility for different BLAS libraries (see Section IV-D). For this reason, the chosen library has to be compiled and linked before its execution.

Typically, a BLAS implementation that has been tuned extensively for performance can be found on HPC machines. The performance of such a BLAS library is difficult to match and not using the provided BLAS will usually not exploit the full potential of a machine. As such, this directly conflicts with the idea of a portable container. We therefore pursue an approach where the container environment does not include an optimized BLAS. It instead provides the software environment to build the application and link it to the host-optimized BLAS available on the host. To this end, the container is built so that it contains the HPL sources, an MPI implementation and the necessary build tools to compile and link the HPL binary. However, the resulting binary is not included in the resulting image, meaning that a build command has to be executed first. The build is performed on the host file system so that the resulting binary environment is persistent. Once the build is successful, a run command can be executed in the container, again specifying the HPL binary path and the `HPL.dat` on the host system.

The options used by the `hpl_build` script that compile and link the containerized HPL are as follows:

```
--linear-algebra-lib <blas library path>     \
--linear-algebra-include <blas header path> \
--linker-option <flags>                      \
--build-path <host fs build directory>
```

The following command shows an example usage for building HPL using the host optimized OpenBlas [16]:

```
shifter                                      \
 --volume=/scratch/linus/fsrootcopy:/hostfs \
 --image=gronerl/hpl-blas-on-host-fs         \
   hpl_build                                 \
--linear-algebra-include /cm/shared/apps/
    openblas/0.2.15/include/openblas/ \
--linear-algebra-lib /cm/shared/apps/openblas
    /0.2.15/lib/libopenblasp64.a \
--build-path /scratch/linus/hpl
```

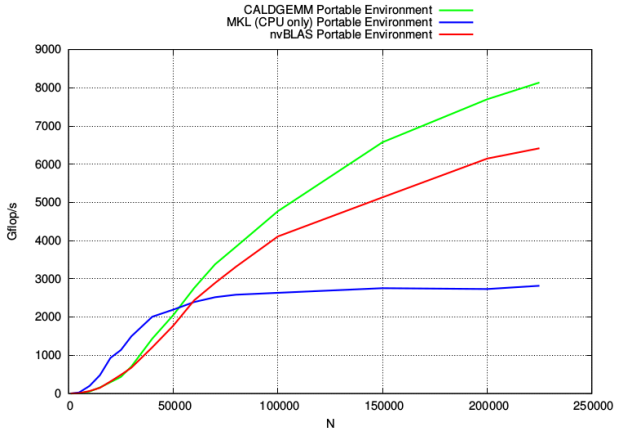

Figure 2. HPL Linpack benchmark running from a Portable Compilation Environment using three different BLAS versions on eight nodes of Piz Daint.

After the compilation phase finished, the same container is used to execute the application. In the following example, the Linpack parameter file should be placed in the same directory than the `xhpl` binary.

```
srun -n 4 -mpi=pmi2 --partition=shifter
    shifter --mpi --volume=/scratch/linus/
    fsrootcopy:/hostfs --image=gronerl/hpl-
    blas-on-host-fs xhpl
```

Figure 2 shows the HPL benchmark results using three different BLAS libraries versions (CALDGEMM [17], MKL [18] and nvBLAS [19]) on eight nodes of Piz Daint. The same portable compilation environment was used for the compilation and later execution of the `xhpl` binary using the procedures shown earlier.

The initial results clearly show how the performance improves depending on the library version used during the compilation-and-linking phase of the portable compilation environment.

V. Conclusion

In this work we described the experiences of building and testing containers with Docker for scientific workflows, while taking advantage of the performance portability provided by Shifter to deploy these containers on HPC infrastructures like Piz Daint. We have covered a variety of scientific-application workflows which can be used as a reference by users who would like to take advantage of the benefits that containers provide.

The performance portability of these workflows, as well as their performance results, support the idea that the proposed Docker-Shifter tool set is a viable alternative for the deployment of non-trivial scientific applications on HPC systems.

Furthermore, the presented workflows describe container creation on a laptop and their deployment at scale on Piz Daint, also taking advantage of the specialized hardware available: Cray Aries interconnect and NVIDIA Tesla P100 GPU accelerators.

Recently, CSCS users have already started using Shifter on Piz Daint and their feedback has so far been on the positive side.

VI. Acknowledgement

The Systems Integration group at CSCS would like to acknowledge Linus Groner's contribution for the implementation and testing of the Portable Compilation Environments presented in Section IV-F.

References

[1] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: a performance comparison," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 386–393.

[2] D. M. Jacobsen and R. S. Canon, "Shifter: Containers for HPC," in *Cray Users Group Conference (CUG'16)*, 2016.

[3] L. Benedicic, F. A. Cruz, A. Madonna, and K. Mariotti, "Portable, high-performance containers for hpc," *arXiv preprint arXiv:1704.03383*, 2017.

[4] Docker, "Docker documentation," available at: https://docs.docker.com/ (March 2017).

[5] Docker, "Base command for the docker cli," available at: https://docs.docker.com/engine/reference/commandline/docker/ (March 2017).

[6] ——, "Docker hub," available at: https://hub.docker.com/ (March 2017).

[7] "Dockerfile for nvidia/cuda:8.0," available at: https://gitlab.com/nvidia/cuda/blob/ubuntu16.04/8.0/runtime/Dockerfile (Mar. 2017).

[8] NVIDIA, "Build and run Docker containers leveraging NVIDIA GPUs," available at: https://github.com/NVIDIA/nvidia-docker (Feb. 2017).

[9] CRAY, "XC Series System Administration Guide (CLE 6.0)," available at: https://pubs.cray.com (Mar. 2017).

[10] Top500.org, "Top500 list - november 2016," available at: https://www.top500.org/list/2016/11/ (March 2017).

[11] L. Nyland, M. Harris, J. Prins *et al.*, "Fast N-Body Simulation with CUDA," *GPU gems*, vol. 3, no. 31, pp. 677–695, 2007.

[12] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[13] Trilinos project, "Trilinos Project Home Page," available at: https://trilinos.org/ (Mar. 2017).

[14] F. D. Witherden, B. C. Vermeire, and P. E. Vincent, "Heterogeneous computing on mixed unstructured grids with PyFR," *Computers & Fluids*, vol. 120, pp. 173–186, 2015.

[15] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[16] "OpenBLAS: An optimized BLAS library," available at: http://www.openblas.net/ (Mar. 2017).

[17] "Portable and Flexible DGEMM Library for GPUs," available at: https://github.com/davidrohr/caldgemm (Mar. 2017).

[18] "Intel Math Kernel Library (Intel MKL)," available at: https://software.intel.com/en-us/intel-mkl (Mar. 2017).

[19] "NVBLAS: GPU-accelerated BLAS implementation," available at: http://docs.nvidia.com/cuda/nvblas/#abstract (Mar. 2017).

APPENDIX

Listing 16. Trillinos `dockerfile`

```
FROM debian:jessie

RUN apt-get update
RUN apt-get install -y build-essential \
                       gfortran        \
                       libopenblas-dev \
                       cmake           \
                       wget            \
                       ca-certificates \
                       --no-install-recommends

# retrieve Trilinos
RUN wget -q https://github.com/trilinos/Trilinos/archive/trilinos-release-12-10-1.tar.gz \
    && tar xf trilinos-release-12-10-1.tar.gz -C /opt                                    \
    && rm -rf trilinos-release-12-10-1.tar.gz
ENV TRILINOS_DIR /opt/Trilinos-trilinos-release-12-10-1

# install MPICH
RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz \
    && tar xf mpich-3.1.4.tar.gz                                           \
    && (cd mpich-3.1.4                                                     \
    && ./configure --disable-fortran --enable-fast=all,O3 --prefix=/usr    \
    && make -j$(nproc)                                                     \
    && make install)                                                       \
    && ldconfig                                                            \
    && rm mpich-3.1.4.tar.gz

# add epetra MPI benchmark (the benchmark developed by sjdeal)
COPY CMakeLists.txt $TRILINOS_DIR/packages/epetra/test/SjdealBenchmark/CMakeLists.txt
COPY cxx_main.cpp $TRILINOS_DIR/packages/epetra/test/SjdealBenchmark/cxx_main.cpp
RUN sed -i $TRILINOS_DIR/packages/epetra/test/CMakeLists.txt -e '/IF (NOT
     Trilinos_NO_32BIT_GLOBAL_INDICES)/aADD_SUBDIRECTORY(SjdealBenchmark)'

# build trilinos + sjdeal's benchmark
RUN mkdir $TRILINOS_DIR/build                  \
    && (cd $TRILINOS_DIR/build                 \
    && cmake                                   \
        -DCMAKE_BUILD_TYPE=RELEASE             \
        -DCMAKE_INSTALL_PREFIX=../install      \
                                               \
        -DTPL_ENABLE_MPI:BOOL=ON               \
        -DMPI_BASE_DIR:PATH=/usr/lib           \
                                               \
        -DTrilinos_ENABLE_OpenMP:BOOL=ON       \
        -DTrilinos_ENABLE_TESTS:BOOL=ON        \
        -DTrilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
        -DTrilinos_ENABLE_Epetra:BOOL=ON       \
        -DTrilinos_ENABLE_CXX11=ON             \
        -DTrilinos_ASSERT_MISSING_PACKAGES=OFF \
        -DBUILD_SHARED_LIBS:BOOL=OFF           \
                                               \
        -DCMAKE_VERBOSE_MAKEFILE:BOOL=OFF      \
        -DTrilinos_VERBOSE_CONFIGURE:BOOL=OFF  \
        ..                                     \
    && make -j$(nproc))
```

Listing 17. PyFR `dockerfile`

```
FROM ubuntu:16.04

LABEL com.pyfr.version="1.5.0"
LABEL com.python.version="3.5"

# Install system dependencies
RUN apt-get update && apt-get install -y   \
        unzip                              \
        wget                               \
        build-essential                    \
        gfortran-5                         \
        strace                             \
        realpath                           \
        libopenblas-dev                    \
        liblapack-dev                      \
        python3-dev                        \
        python3-setuptools                 \
        python3-pip                        \
        libhdf5-dev                        \
        libmetis-dev                       \
        --no-install-recommends         && \
    rm -rf /var/lib/apt/lists/*


# Label image so that nvidia-docker will source driver libraries from host
LABEL com.nvidia.volumes.needed="nvidia_driver"

# Install CUDA Toolkit 8.0
ENV CUDA_VERSION 8.0
LABEL com.nvidia.cuda.version="8.0"

RUN NVIDIA_GPGKEY_SUM=d1be581509378368edeec8c1eb2958702feedf3bc3d17011adbf24efacce4ab5 && \
    NVIDIA_GPGKEY_FPR=ae09fe4bbd223a84b2ccfce3f60f4b3d7fa2af80                          && \
    apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/
        ubuntu1604/x86_64/7fa2af80.pub && \
    apt-key adv --export --no-emit-version -a $NVIDIA_GPGKEY_FPR | tail -n +5 > cudasign.pub
        && \
    echo "$NVIDIA_GPGKEY_SUM  cudasign.pub" | sha256sum -c --strict - && rm cudasign.pub && \
    echo "deb http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1604/x86_64 /" >
        /etc/apt/sources.list.d/cuda.list

ENV CUDA_PKG_VERSION 8-0=8.0.44-1
RUN apt-get update && apt-get install -y --no-install-recommends \
      cuda-core-$CUDA_PKG_VERSION                                \
      cuda-misc-headers-$CUDA_PKG_VERSION                        \
      cuda-command-line-tools-$CUDA_PKG_VERSION                  \
      cuda-cublas-dev-$CUDA_PKG_VERSION                          \
      cuda-curand-dev-$CUDA_PKG_VERSION                          \
      cuda-cudart-dev-$CUDA_PKG_VERSION                          \
      cuda-driver-dev-$CUDA_PKG_VERSION                       && \
    ln -s cuda-$CUDA_VERSION /usr/local/cuda                  && \
    rm -rf /var/lib/apt/lists/*

RUN echo "/usr/local/cuda/lib64" >> /etc/ld.so.conf.d/cuda.conf   && \
    ldconfig

RUN echo "/usr/local/nvidia/lib" >> /etc/ld.so.conf.d/nvidia.conf    && \
    echo "/usr/local/nvidia/lib64" >> /etc/ld.so.conf.d/nvidia.conf

ENV PATH /usr/local/nvidia/bin:/usr/local/cuda/bin:${PATH}
ENV LD_LIBRARY_PATH /usr/local/nvidia/lib:/usr/local/nvidia/lib64${LD_LIBRARY_PATH}

# Install MPICH 3.1.4
```

```
RUN wget -q http://www.mpich.org/static/downloads/3.1.4/mpich-3.1.4.tar.gz  && \
    tar xvf mpich-3.1.4.tar.gz                                              && \
    cd mpich-3.1.4                                                          && \
    ./configure --disable-fortran --prefix=/usr                             && \
    make -j$(nproc)                                                         && \
    make install                                                            && \
    cd ..                                                                   && \
    rm -rf mpich-3.1.4.tar.gz mpich-3.1.4                                   && \
    ldconfig

# Create new user
RUN useradd docker
WORKDIR /home/docker

# Install Python dependencies
RUN pip3 install numpy>=1.8        \
                 pytools>=2016.2.1  \
                 mako>=1.0.0        \
                 mpi4py>=2.0     && \
   pip3 install pycuda>=2015.1     \
                 h5py>=2.6.0     && \
   wget -q -O GiMMiK-2.1.tar.gz https://github.com/vincentlab/GiMMiK/archive/v2.1.tar.gz && \
   tar -xvzf GiMMiK-2.1.tar.gz  && \
   cd GiMMiK-2.1                && \
   python3 setup.py install     && \
   cd ..                        && \
   rm -rf GiMMiK-2.1.tar.gz GiMMiK-2.1

# Set base directory for pyCUDA cache
ENV XDG_CACHE_HOME /tmp

# Install PyFR
RUN wget -q -O PyFR-1.5.0.zip http://www.pyfr.org/download/PyFR-1.5.0.zip && \
    unzip -qq PyFR-1.5.0.zip                                              && \
    cd PyFR-1.5.0                                                         && \
    python3 setup.py install                                              && \
    cd ..                                                                 && \
    rm -rf PyFR-1.5.0.zip PyFR-1.5.0
```