Cray Programming Environments within containers on Cray XC systems

Maxime Martinasso, Miguel Gila, William Sawyer, Rafael Sarmiento, Guilherme Peretti-Pezzi, Vasileios Karakasis
Swiss National Supercomputing Centre, ETH Zürich, Lugano, Switzerland
{firstname}.{lastname}@cscs.ch

***Abstract*—We present a methodology to enable the complete software development life cycle on Cray XC systems within a container which can hold any version of the Cray Programming Environment (CPE). The installation of the CPE inside a container facilitates many aspects of the typical HPC support and operation workloads of managing Cray XC systems such as testing new CPEs, comparing CPE performances or keeping software built with an old CPE running on updated systems. The procedure for creating a container with a CPE inside consists of three steps: The creation of a container holding the targeted CPE, the compilation of the desired software within such containers, and the packaging of the resulting binaries, libraries and dependencies within a second lightweight container. We show case the methodology by fulfilling a user requirement of running a 2-years old version of the COSMO model built with an old CPE 16.11 on today's system.**

***Keywords*-Containers; Reproducible experiments; Cray Programming Environment;**

I. Introduction

Container technologies have received a great deal of attention over the recent years and arguably have changed the way software is deployed and packaged. Thanks to the Linux kernel capabilities, containers were initially developed to provide isolation of processes at runtime. Lately, containers have been used to achieve many interesting objectives, which would be otherwise cumbersome to attain, like the portability and the packaging of systems and software stacks in order to enable the possibility of seamlessly running software on different environments.

In the High-Performance Computing (HPC) community, containers have been welcomed mainly as a way to package software stacks into supercomputer facilities and to manage large ecosystems of interdependent applications. Container frameworks [1] [2] have been developed specifically for HPC to fulfill requirements such as bringing native accelerator performance inside the container or disabling the usage of root to prevent privileged escalation on a shared file system. However, the exploitation of containers on HPC systems is still in an early stage, and it is foreseeable that new use cases will come to benefit from their many advantages. For instance, on the side of HPC users, one often finds the use of containers as a tool for ensuring portability of applications and reproducible research [3].

For managing HPC applications ecosystem, containers are useful where changes to the programming environment (PE) may affect both performance and the calculation results of applications. Although easy to spot through regression tests, such issues typically are very hard to solve, in particular on complex programming environments, where applications have many dependencies including a number of scientific libraries and system-specific libraries such as the CUDA Toolkit in the case of system with Nvidia GPUs. Such performance and variability in results arise on a production system when CPEs are updated and user expectations need to be addressed. For instance on the Cray XC systems which are targetted in this paper, CPE updates generally involve completely rebuilding the supported scientific libraries and applications. These updates often can be disruptive to the applications or workflows directly maintained by the end users.

In this work we present a methodology to containerize any version of the CPE in order to enable the complete software development life cycle on Cray XC systems within containers. A container with the CPE inside offers the possibility of testing an application's deployment and performance on upcoming CPE versions ahead of system updates, thus minimizing the need of changes in the physical systems (test or production) and reducing the time required to test new releases. Similarly, CPE in containers allows old and recent CPE versions to co-exist in the same system with lower installation and maintenance effort without affecting software already installed and running in production. Our methodology allows the design of more powerful continuous integration pipelines and a more efficient planning of system updates, as well as granting more resilience to the transitions between CPE versions.

The key contributions of our work are:

- to create a container holding any version of a Cray Programming Environment;
- to explain the usage of such container to build and to run HPC applications;
- to showcase the usefulness of containerized CPE on a concrete use case based on a complex and difficult-to-build scientific application.

II. Cray Programming Environment inside a container

Our methodology is based on building a CPE inside a container image (or CPE container) and then using bespoke container as an environment for software development and maintenance. It consists of three main parts: the creation of a container image holding a CPE, the compilation of software within such container, and the packaging of the resulting binaries, libraries and dependencies in general within lightweight container images.

A. Creating containers holding the CPE

Before building a CPE container we need to setup directories on the local machine (which has Docker installed, like a laptop) to hold the CPE and related packages. The CPE is downloaded from CrayPort[1] as an ISO file. The ISO file is named after a Cray Development Toolkit image (CDT), and we mount it locally on a directory named after its version (such as `volume/CDT-18.10-03PRE`). To enable the CUDA Toolkit, we also copy Cray-provided CUDA packages to a local directory named `cuda`.

The main step consists of building a Docker [4] container which holds the CPE. The Listing 2 shows the `Dockerfile` used to create a CPE container.

As a base image for this container, we use a Cray eLogin image of Piz Daint, and we convert it to a container image with the help of SquashFS tools. This eLogin image contains the same Cray Linux Environment available on any eLogin node, without any CPE installed yet. Such a newly created container image is used as a base image denoted by the `FROM` keyword.

Then we set build arguments to default values. These arguments are used to select the CPE version and other related required parameters. One can change their value at the command line when building the container with the requirement that the corresponding version of the CDT should be mounted in the local directories together with the CUDA packages. Then we setup a user `pe_user` who will be used to build software packages inside the CPE container.

The next step on the `Dockerfile` is to copy the content of `volume` and `cuda` folders which holds the Cray packages to install. Copying these files can be avoided by using the `--mount` option of `docker run` command. However, this command works only with Docker version 17.06 and newer, and as per today it is not a generic solution.

Once the packages are copied, we follow the Cray-supported procedure of installing a CPE [5]. We first setup a configuration file with the aforementioned build argument values and then execute a Cray-provided script to install the CPE. Extra packages such as the CUDA Toolkits are also installed in that step. Cray packages are provided as `RPM`. We have observed that some CDT versions don't always install all required packages. In that case, an extra line is added to install the missing packages making the `Dockerfile` specific to a version of a CDT.

Finally we set the container for the `pe_user` and we setup his environment (`MODULEPATH`) to enable access of CPE through the modules system. Once inside the container, the environment is identical to the one of a Cray HPC platform. The procedure to build a binary inside the CPE container is identical to the one used on a Cray XC platform.

The whole process to create a CPE container takes a couple of hours depending on the local machine and the size of the CDT ISO file. The size of the resulting container image is rather large (several tens of GB), which might be not convenient to manipulate, deploy or copy. Therefore, such image is not suitable to execute on a HPC platform, and binaries built with a CPE container should be packaged differently.

To summarize, Listing 1 displays the list of commands required to build a CPE container.

CDT version	ISO file size	CPE container size
16.11-07	4.5 GB	34 GB
17.08-06	4.2 GB	32 GB
18.10-03	6.1 GB	50 GB

Table I
SIZE OF ISO FILE AND CPE CONTAINERS FOR VARIOUS VERSION OF CDT (INCLUDING THE CUDATOOLKIT PACKAGE).

B. Building software within a CPE container

Once the image with the CPE is ready, it is possible to build applications within it. This can be done by running the container interactively or by creating a new `Dockerfile` with `RUN` statements. In any case, we recommend to use the container interactively beforehand to setup a packaging tool such as EasyBuild [6] or Spack [7].

As mentioned before, due to the CPE container size, it is not practical and useful to bundle the CPE container with the software being built. Table I presents the different sizes of CPE container versus their ISO file sizes. Therefore, we create two directories in the CPE container, the first one serves as recipient of the source code (named input directory in the rest of the paper), and the second one contains the generated binaries after compilation (named output directory). These directories are mounted from the local machine into the container.

C. Packaging binaries and dependencies on a lightweight image

At this point, the applications are built and accessible from outside the CPE container. However, many dependencies of the application are referring to libraries inside the CPE container, preventing the software to run from outside the container. In order to copy the required CPE dependencies into the output directory, we created a python script (`ldd_parser`). This script uses a combination of Linux tools such as `ldd` and `strings` to identify the dependencies. The script is applied recursively on all dependencies and returns a list of libraries. Common Linux li-

[1]htpp://crayport.cray.com

Listing 1. List of commands to build and to run a CPE container

```
# build and import the base image
$ sudo unsquashfs -f -d unsquashfs elogin_prod_up07_20181205160931.squashfs
$ sudo tar -C unsquashfs -c . | docker import - elogin_prod:up07_20181205160931

#directory structure
$ ls .
Dockerfile
cuda
volume
$ ls cuda/
cray-cudatoolkit7.5-7.5.18_2.1.7-6.0.7.0_18.1__gd80efc5.x86_64.rpm
cray-cudatoolkit8.0-8.0.61_2.4.9-6.0.7.0_17.1__g899857c.x86_64.rpm
cray-cudatoolkit9.0-9.0.103_3.15-6.0.7.0_14.1__ge802626.x86_64.rpm
cray-cudatoolkit9.1-9.1.85_3.18-6.0.7.0_5.1__g2eb7c52.x86_64.rpm
cray-cudatoolkit9.2-9.2.148_3.19-6.0.7.1_2.1__g3d9acc8.x86_64.rpm
cray-nvidia-libcuda-390.46_3.1.30-6.0.7.0_24.8__g83596c3.ari.x86_64.rpm
cray-nvidia-libcuda-390.46_3.1.32-6.0.7.1_6.1__g15a0cc2.ari.x86_64.rpm
cray-nvidia-libcuda-396.44_3.1.31-6.0.7.1_2.1__g97ab0cf.ari.x86_64.rpm
cray-nvidia-libcuda-396.44_3.1.33-6.0.7.1_3.2__gac01daf.ari.x86_64.rpm
$ ls volume/16.11-07/
TRANS.TBL conf docs installer packages release_info

# build the CPE container
$ docker build --build-arg CDT_VERSION=16.11-07 --build-arg CUDATOOLKIT=8.0 \
            -t craype:cdt16.11-07.haswell.pascal.cudatoolkit8.0 .

 # start the container interactively
 $ docker run -v /Users/maximem/dev/docker/my_source:/home/pe_user/sources \
          -v /Users/maximem/dev/pe_container/my_binaries:/home/pe_user/install \
          --rm -it craype:cdt16.11-07.haswell.pascal.cudatoolkit8.0 /bin/bash
```

braries like `libpthread.so`, `libdl.so` or `librt.so` are discarded as they might conflict with the ones that are installed on the Cray system.

For complex applications, one major difficulty is to identify libraries that are dynamically opened inside the code by the use of the `dlsym` mechanism. To help in that task, the script will parse the text section of the binaries by using the `strings` command to identify the usage of `dlsym` and to find the names of the loaded libraries. Listing 3 shows an output of the script. To run the application and find all the libraries at runtime, it is then necessary to preempt the `LD_LIBRARY_PATH` environment variable with the folder which holds all the dependencies. Thus, binaries compiled for a specific CPE can be executed on Cray system without having that specific CPE installed.

Instead of copying the output directory to the XC system where the binaries will be run and to manually set up the environment, an alternative approach is to bundle the binaries and their dependencies within a lightweight container image. This can be done with a simple `Dockerfile` which may use any light Linux distribution as a base image. Doing this, applications can be shipped on images which typically take only hundreds of MB.

III. Limitations and best practices

By using this methodology to build a large number of CPE containers we have identified a set of limitations and best practices:

- As mentioned previously, the Cray-provided installation system is not robust over all CDT ISO images. We have seen that some packages are not installed by default (even the Cray compiler) in certain CDTs. Nevertheless, recent CDT versions seem more consistent. As a solution, one can start the CPE container as `root` and install the missing packages. By switching from `root` to the `pe_user` one can easily progress to build the required software by installing the missing packages. Finally, these missing commands can be integrated into a CDT version-specific `Dockerfile`.
- A CPE in a container raises the question of the right to distribute the CPE. Already today any user on a system could copy part of the CPE outside of the HPC facility, CPE containers accentuate that possibility. After informing Cray about this work initiative and asking about the possibility to use a CPE in a container, they stipulated that such container must be executed on a targeted Cray machine associated to the downloaded ISO file. However, Cray allows to prepare container images and building codes that target a specific Cray (for example Piz Daint) offline from the Cray machine.
- CSCS intention is not to provide such CPE container capability as a service for the end users but instead to utilize it as an internal service for well-defined use cases: regression testing for upgrades, reproducibility experiments or performance analysis across CPE versions. One requirement is to provide a server with large disk capacity on which Docker is running inside a virtual machine for security reasons.
- The `ldd_parser` script is helpful to identify dependencies but is not a completely reliable solution.

Listing 2. CPE container Dockerfile

```
FROM elogin_prod:up07_20181205160931
ARG CDT_VERSION=18.10-03PRE
ARG CPU_TARGET=haswell
ARG ACCELERATORS=PASCAL
ARG CUDATOOLKIT=9.2

# Setup directories and pe user
RUN mkdir /root/${CDT_VERSION} && \
   mkdir /root/cuda && \
   mkdir /root/logs && \
   useradd -ms /bin/bash pe_user && \
   mkdir /home/pe_user/sources && \
   mkdir -p /home/pe_user/install/craype_runtime && \
   echo "CrayPe Version: cdt:${CDT_VERSION} cpu:${CPU_TARGET} acc:${ACCELERATORS} cudatoolkit:${CUDATOOLKIT}"\
       > /home/pe_user/install/craype_runtime/craype_version.txt

COPY volume/${CDT_VERSION} /root/${CDT_VERSION}

COPY cuda/ /root/cuda/

# One could use the mount option of `RUN` to avoid the copy but that works only
# with specific version of docker
#RUN --mount=target=/root/${CDT_VERSION},type=bind,source=volume/${CDT_VERSION} \

# Edit configuration and install packages
RUN cd /root/${CDT_VERSION}/installer && \
   rpm -ivh craype-installer-*.rpm --upgrade && \
   cp /opt/cray/craype-installer/default/conf/install-cdt.yaml /root && \
   sed -i -e "s/LOGS_DIR[[:space:]]*:[[:space:]]*NEED-TO-SPECIFY/LOGS_DIR : \/root\/logs/" \
        -e "s/ISO_MOUNT_DIR[[:space:]]*:[[:space:]]*NEED-TO-SPECIFY/ISO_MOUNT_DIR : \/root\/${CDT_VERSION}/" \
        -e "s/INSTALL_PGI_LIBRARIES[[:space:]]*:[[:space:]]*NO/INSTALL_PGI_LIBRARIES : YES/" \
        -e "s/INSTALL_INTEL_LIBRARIES[[:space:]]*:[[:space:]]*NO/INSTALL_INTEL_LIBRARIES : YES/" \
        -e "s/CRAY_CPU_TARGET[[:space:]]*:[[:space:]]*NEED-TO-SPECIFY/CRAY_CPU_TARGET : ${CPU_TARGET}/" \
        -e "s/ACCELERATORS[[:space:]]*:[[:space:]]*NONE/ACCELERATORS : ${ACCELERATORS}/" /root/install-cdt.yaml && \
   cd /root/ && \
   /opt/cray/craype-installer/default/bin/craype-installer.pl --install --install-yaml-path install-cdt.yaml \
                                           --network ari && \
   rpm -ivh /root/cuda/cray-cudatoolkit${CUDATOOLKIT}-*.rpm \
         /root/cuda/cray-nvidia-libcuda-396.44_3.1.33-6.0.7.1_3.2__gac01daf.ari.x86_64.rpm

USER pe_user
WORKDIR /home/pe_user/sources

ENV MODULEPATH /opt/cray/pe/perftools/default/modulefiles:/opt/cray/pe/craype/default/modulefiles:
           /opt/cray/pe/modulefiles:/opt/cray/modulefiles:/opt/modulefiles:/opt/cray/ari/modulefiles:
           /opt/cray/craype/default/modulefiles
```

Listing 3. Output of the ldd_parser script

```
$ ldd_parser --binaries /opt/cray/pe/cce/8.7.3/cce/x86_64/lib/libfi.so
/opt/cray/pe/gcc-libs/libstdc++.so.6
/opt/cray/pe/gcc-libs/libgfortran.so.3
/opt/cray/pe/cce/8.7.3/cce/x86_64/lib/libf.so.1
/opt/cray/pe/cce/8.7.3/cce/x86_64/lib/libcsup.so.1
/opt/cray/pe/cce/8.7.3/cce/x86_64/lib/libu.so.1
/opt/cray/pe/cce/8.7.3/cce/x86_64/lib/libcraymath.so.1
# Duplicated reference of libraries:
/opt/cray/pe/cce/8.7.3/cce/x86_64/lib/libquadmath.so.0
/opt/cray/pe/gcc-libs/libgcc_s.so.1
/opt/gcc/6.2.0/snos/lib/../lib64/libgcc_s.so.1
/opt/gcc/6.2.0/snos/lib/../lib64/libquadmath.so.0
# dlsym libraries:
libmemkind.so.0
libnuma.so
```

A manual intervention is unavoidable, especially for libraries opened with `dlopen`, whose locations are not known at compilation time.

- EasyBuild or any package manager should not be installed inside the container but rather be accessed from the input directory. In that way, the container does not need to be re-created for a version change of the package manager or an update of the list of recipes.

IV. Use case: Reproducible experiments

The COSMO [8] (Consortium for Small-scale Modeling) model for regional numerical weather forecasting is complex scientific applications widely used in the climate and weather community. It is run daily by different institutions in the world like MeteoSwiss, and, it is also used at climate research departments such as the Institute for Atmospheric and Climate Science at the Swiss Federal Institute of Technology (IAC-ETHZ) to explore climate effects.

COSMO is the first weather and climate application that has been ported to GPU. It consists of different components written in different languages such as the physics computation in Fortran with OpenACC directives and the atmospheric dynamics solver using C++ and the CUDA library. These components are built and combined together in a single binary by using a complex Cmake build system.

For climate researchers it is important to ensure reproducible experiments over 2 to 4 years time due to publication requirements. To that end, IAC-ETHZ has asked CSCS to help them to ensure such reproducible experiments on Piz Daint. Their current procedure is to build the same (outdated) COSMO version for every update of the CPE on the Piz Daint systems. It becomes for them time-consuming activity to successfully build and run their COSMO version with each new CPE. Another large portion of their time is dedicated to re-validate the obtained scientific results.

To be more specific, IAC-ETHZ requires the COMSO-OPCODE version of COSMO which has had very few commits since 2015 (mostly to fix CPE update issues), built with the CDT 16.11-07 for which it has been validated. CDT 16.11-07 has been released around November 2016. Today Piz Daint has CDT 18.09 installed and soon CDT 19.03. While several CDT versions can be installed in the same system, the sheer size of the CDT and the subsequent CPE image projected to the compute and elogin nodes makes it impractical to install every single version available. For reference, the current CPE image on Piz Daint is in the order of 140GB with 5 CDTs installed.

In order to enable reproducible experiments with COSMO-OPCODE, we have built a CPE container with version CDT 16.11-07 and the CUDA Toolkit 8.0. Once the container was created, we successfully built COSMO-OPCODE following the build instructions. This task was not straightforward and it led to a list of missing packages to install and to changes in the environment. By creating a lightweight container and identifying all the dependencies (35 libraries) of the COSMO-OPCODE built, we were able to run the regression of COSMO-OPCODE on Piz Daint. All test passed.

V. Use case: Performance over CPE versions

In this use case, we are investigating the performance variability of software compiled with different CPEs. As a target application, we have selected the application CP2K [9] which is a Quantum Chemistry and Solid-state Physics software package. CP2K is commonly used by the HPC community and it has been ported to GPUs [10]. We have set CP2K to the version 6.1.

Thanks to our methodology, we have created two CPE containers, one using a CDT version 17.08 (July 2017) with the CUDA Toolkit 8.0 and one using a more recent version: CDT version 18.08 (July 2018) with the CUDA Toolkit 9.1. We have compiled CP2K inside those containers and extracted the binaries and dependencies. Our target platform is Piz Daint and both versions of CP2K have been executed on different number of nodes. We used the model `H2O-256` which is part of CP2K's set of benchmark models.

Figure 1 presents the performance values of both CP2K versions. For a small number of nodes, performance is similar for both CDTs, however, for a larger number of nodes, CDT 17.08 seems to perform around 10% better than CDT 18.08. This result confirms that upgrading a CPE does not necessarily imply better performance.

The CPE container methodology allows us to build our set of reference applications with different CDT ISO images and to evaluate their performance. By doing so, we can, on the one hand, investigate performance discrepancies, and, on the other hand, manage user expectations. It also becomes interesting to keep an application built with a specific CPE if it runs faster.

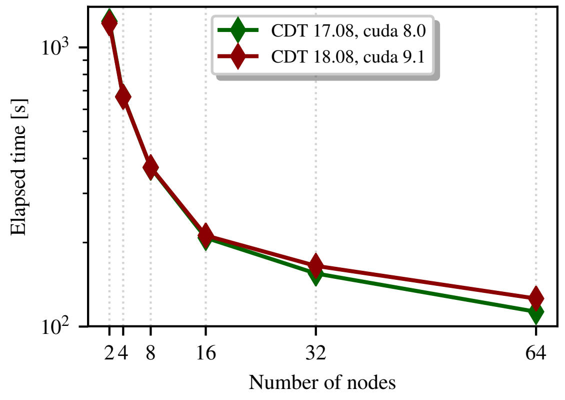

Figure 1. Performance comparison of CP2K, model H20-256. CDT 17.08 with CUDA 8.0 performs 10% better at 64 nodes than a most recent CDT 18.08 with CUDA 9.1.

VI. Related work

Usage of containers comes from the Enterprise and microservices community. This community has developed a large experience in building software using containers. For instance, they have introduced the *twelve factor apps* methodology [11] for which containers fit naturally [12]. One of the principles of this methodology is to separate the build and run phase when developing and deploying software. This motivates us to deploy applications outside of a CPE container.

Today, this community remains very active. It provides very advanced tools for the automation of building software inside containers. For instance, Binder [13] creates Jupyter notebooks inside containers from a github repository. This allows applications to be immediately reproducible by anyone, anywhere. Such tool could be investigated to help reproducible experiments in the HPC community.

Containers in general are being studied for scientific reproducibility experiments [14] [3]. Docker has proven to simplify outstanding issues for these experiments, for example, software dependencies, software version management and software distribution.

VII. Future work

In this work, we have developed a methodology to containerize any CPE version. A CPE container allows the recreation of a Cray environment on any machine and to compile any application with it. Once built, an application is packaged on a lightweight container and run on an HPC system. As we have presented in this paper, this methodology could be used to ensure portability of code over different CPE updates on a system for improving the capability to reproduce scientific experiments. It also allows the comparison of performance variation over CPE versions of key HPC applications.

As a future work, we will use the CPE container methodology to test and validate new CPEs before they are installed on a system. This will allow us to capture early on issues and to start pro-active discussion with Cray instead of a being in a reactive situation where more minor updates of CPE are installed to fix user issues. We will integrate such methodology in our tool sets to automate testing over different CPE versions.

All the procedure and scripts are available on request to CSCS. We plan to open source this work in a near future.

References

[1] L. Gerhardt, W. Bhimji, S. Canon, M. Fasel, D. Jacobsen, M. Mustafa, J. Porter, and V. Tsulaia, "Shifter: Containers for HPC," *Journal of Physics: Conference Series*, vol. 898, p. 082021, 10 2017.

[2] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PLOS ONE*, vol. 12, no. 5, pp. 1–20, 05 2017. [Online]. Available: https://doi.org/10.1371/journal.pone.0177459

[3] C. Boettiger, "An introduction to Docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[4] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[5] Cray Inc., "XC series software installation and configuration guide (CLE 7.0.UP00) s-2559 rev c," https://pubs.cray.com/content/S-2559, 2019-03-08.

[6] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt, "EasyBuild: Building software with ease," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 572–582. [Online]. Available: https://doi.org/10.1109/SC.Companion.2012.81

[7] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: bringing order to HPC software chaos," in *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2015, pp. 1–12. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/2807591.2807623

[8] B. Rockel, A. Will, and A. Hense, "The regional climate model COSMO-CLM (CCLM)," *Meteorologische Zeitschrift*, vol. 17, no. 4, pp. 347–348, 2008.

[9] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "CP2K: atomistic simulations of condensed matter systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/wcms.1159

[10] O. Schütt, P. Messmer, J. Hutter, and J. VandeVondele, *GPU-Accelerated Sparse Matrix-Matrix Multiplication for Linear Scaling Density Functional Theory*. John Wiley & Sons, Ltd, 2016, ch. 8, pp. 173–190. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781118670712.ch8

[11] A. Wiggins, "The twelve-factor app," *The Twelve-Factor App*, 2011.

[12] K. Matthias and S. P. Kane, *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly Media, Inc., 2015.

[13] Jupyter Project, M. Bussonnier, J. Forde, J. Freeman, B. Granger, T. Head, C. Holdgraf, K. Kelley, G. Nalvarte, A. Osheroff, M. Pacer, Y. Panda, F. Perez, B. Ragan-Kelley, and C. Willing, "Binder 2.0-reproducible, interactive, sharable environments for science at scale," in *Proceedings of the 17th Python in Science Conference*, 2018, pp. 113–120.

[14] R. Chamberlain and J. Schommer, "Using Docker to support reproducible research," *DOI: https://doi.org/10.6084/m9.figshare.1101910*, 2014.