Deploying Alternative User Environments on Alps

Jonathan Coles
CSCS
Zurich, Switzerland
jonathan.coles@cscs.ch

Theofilos-Ioannis Manitaras
CSCS
Lugano, Switzerland
theofilos.manitaras@cscs.ch

Simon Pintarelli
CSCS
Zurich, Switzerland
simon.pintarelli@cscs.ch

Ben Cumming
CSCS
Zurich, Switzerland
bcumming@cscs.ch

Jean-Guillaume Piccinali
CSCS
Lugano, Switzerland
jgp@cscs.ch

Harmen Stoppels
Stoppels Consulting/LLNL
Zurich, Switzerland
harmen@stoppels.ch

***Abstract*—We describe a method for defining, building and deploying alternative programming environments alongside the CPE on HPE Cray EX Alps infrastructure at CSCS. This addresses an important strategic need at CSCS to deliver tailored environments within our versatile cluster (vCluster) configuration. We provide compact, testable, optimized software environments that can be updated independently of the CPE release cycle. The environments are defined with a descriptive YAML recipe, which is processed by a novel configuration tool that builds the software stack using Spack and generates a SquashFS image. Cray-MPICH is provided through a custom Spack package without the need for a CPE installation. We describe the command line tools and Slurm plugin that facilitate loading environments per user and per job. Through a series of benchmarks we demonstrate application and micro-benchmark performance that matches CPE.**

***Index Terms*—CPE, squashfs, slurm, spack**

I. Introduction

CSCS is deploying logically isolated, versatile software-defined clusters (vClusters) [3] on the HPE Cray EX system Alps, to provide services to a wider range of user domains, each with their own software, security, reliability and scaling requirements. The vClusters can be customized for each target use case, as an alternative to one large system that offers a "one size fits all" programming environment, storage and job scheduler configuration.

CSCS aims to reduce the complexity of the software installed on each vCluster through tailored user environments (uenv's). These environments will provide the smallest possible set of compilers, libraries and tools optimized for vCluster's requirements, node architecture and the Slingshot 11 interconnect. One obvious use case is for single purpose clusters, e.g., the production cluster of the Swiss weather service. Alternatively, multiple use-case specific environments can be provided on general-purpose HPC vClusters, and loaded according to a user's individual needs.

This approach is at odds with the widely-adopted method to provide software on HPE Cray EX systems: installing the Cray Programming Environment (CPE), and building use-case specific software not provided by CPE on a shared file system, as illustrated in Fig. 1 (a). The CPE provides a wide range of software – compilers, scientific libraries, communication libraries, debuggers, profiling tools, etc. – all optimised for the node and network architecture of the system. Furthemore, Cray continues to evolve and expand the CPE in response to changing requirements, for example adding software packages for ML/AI, with quarterly releases to address bugs, improve performance and add features. Indeed, CSCS currently delivers such a one-size-fits-all environment, similarly to other HPC sites, using EasyBuild to provide additional software built and maintained by CSCS [4].

However, while the CPE is a good general purpose environment for users, using it as a layer in HPC software stacks conflicts with our aim of reducing the complexity of software stacks. In particular, two issues arise:

- No single use-case or domain will use more than a small subset of the features provided by the CPE;
- Due to the CPE's quarterly cycle, the lead time between identifying an issue and a fix available and tested on site can be expected to be in the order of 3-6 months;

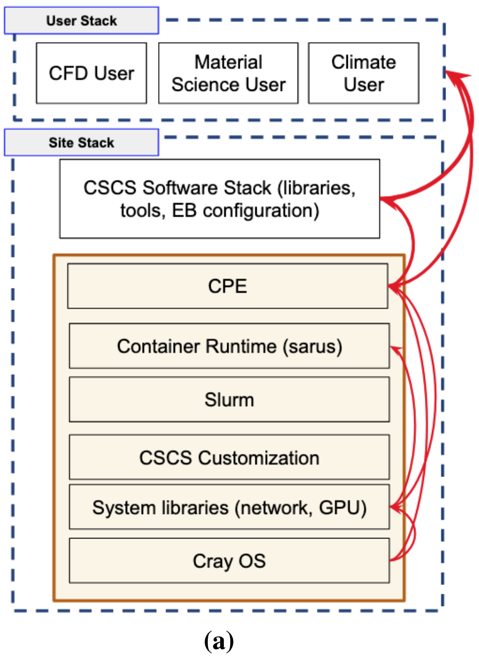
By striking a balance between long term stability and providing up-to-date software versions, CPE cannot fully satisfy use cases that only require either stability or timely fixes.

The work presented in this paper uses Spack and SquashFS images to build and deploy software stacks on top of a simplified base image that provides only the necessary vendor-specific libraries, for example libfabric, as illustrated in Fig. 1 (b). Such a base image changes less frequently than CPE, reducing the need to rebuild software stacks with each CPE update and reducing system dependencies that could require intervention.

The result is compact, testable, reproducible and optimized software environments based on a descriptive recipe that can be updated independently of the CPE release cycle.

II. Spack Stacks

This section introduces a workflow and tooling for building use-case-specific programming environment (PE) stacks on top of a base node image with CrayOS and core dependencies such as libfabric and Slurm, that does not require the CPE.


User Stack
CFD User
Material Science User
Climate User
Site Stack
CSCS Software Stack (libraries, tools, EB configuration)
CPE
Container Runtime (sarus)
Slurm
CSCS Customization
System libraries (network, GPU)
Cray OS
(a)

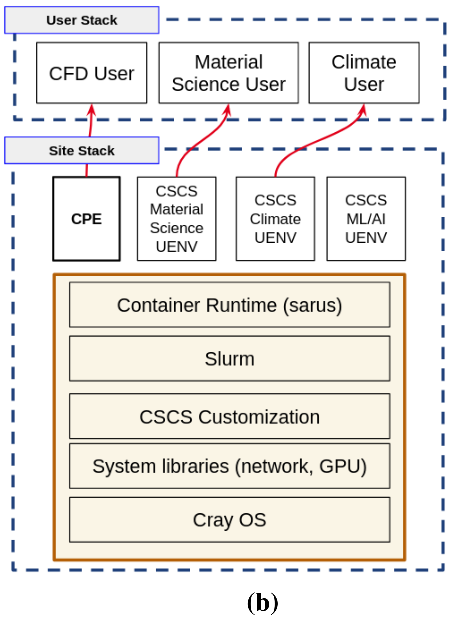


Fig. 1: **(a)**: The "standard" HPE-EX software stack, with the Cray OS, drivers, CPE and site-specific software in the system image, site-provided software installed on a shared file system. User-installed software depends on the software layers underneath. The red arrows indicate where changes to one layer have a knock on effect on other software layers, requiring rebuilds or reconfiguration.
(b): The Alps software stack with the programming environment moved out of the base image. Multiple programming environments can be deployed on top of this architecture. The CPE, and alternative PEs discussed in Section II, are mounted at runtime in a new mount namespace by users.

The main tool is Stackinator[1], which uses Spack [5] to build a complete PE stack in a directory, which can then be deployed as SquashFS images or as directories on a shared file system. Stackinator is an open-source Python application, that is *opinionated* for targeting the vClusters on Alps[2], in the sense that it:

- makes design decisions that focus on reproducability and performance tuning for the target SlingShot 11 network and node architectures available on Alps;
- provides limited configuration options for compilers – the tool configures the full compiler specification according to CSCS best practices;
- and provides limited configuration options for MPI – only Cray-MPICH is fully supported (with future support for MPICH and OpenMPI planned) and for which the tool will configure for Slingshot 11 and accelerator compatibility.

The following sections describe the workflow, from recipe to squashfs-images, and the novel and HPE Cray EX-specific implementation details.

[1]https://github.com/eth-cscs/stackinator
[2]Minimal changes are required to generate stacks for other systems – a version that generates stacks for AWS Gravitron 3 clusters was created in a hackathon.

A. Stack Specification

Stackinator generates stacks from a descriptive YAML file recipe. To explain Spack recipes, we will work with an example stack for development on Cray EX EX235n nodes with NVIDIA A100 GPUs and AMD Zen3 Epyc CPUs. The stack supports development of GPU-aware MPI applications with both GCC and CUDA, and applications with NVIDIA HPC SDK using OpenACC. It is a stripped down version of a software stack that CSCS provides to the Swiss National Weather Service (MeteoSwiss) on Alps, specifically:

- A GCC 11.3 compiler tool chain.
- An NVHPC 22.7 compiler tool chain.
- A GCC programming environment `prgenv-gcc` with CUDA-aware Cray-MPICH, OSU Benchmarks, OpenBLAS and CUDA 11.8.
- An NVHPC programming environment `prgenv-openacc` with CUDA-aware Cray-MPICH, OSU Benchmarks and CUDA 11.8.

The Stackinator recipe is composed of the following files, stored in a common path:

- `config.yaml` specifies where the image will be installed / mounted (the CSCS default is */user-environment*), the version of Spack and optional configuration for a Spack build cache.

```
name: arbor-dev
store: /user-environment
system: hohgant
spack:
  repo: https://github.com/spack/spack.git
  commit: releases/v0.19
mirror:
  enable: false
```

Reproducible builds use a release branch / version tag a specific commit of Spack. A rolling release can be configured by using the `develop` branch of Spack, which will build with the most recent Spack recipes.

- `compilers.yaml` describes the compiler tool chains that the stack provides:

```
bootstrap:
  spec: gcc@11
gcc:
  specs:
  - gcc@11.3
llvm:
  requires: gcc@11.3
  specs:
  - nvhpc@22.7
```

The bootstrap version of GCC built using the system compiler (GCC 7.5 at the time of writing) is not provided as a module or part of the Spack upstream presented to users, instead it is used to build the subsequent GCC compiler toolchains. This step is required to ensure that GCC is built using a compiler that can generate instructions optimised for the target Zen2 and Zen3 micro-architecture CPUs. It is also mandatory to specify at least one version of GCC, in this case GCC 11.3, which is the highest version compatible with CUDA 11. The LLVM tool chains are optional, with support for installing multiple versions of the NVIDIA HPC-SDK and LLVM/Clang.

- `environment.yaml` describes the software packages:

```
prgenv-gcc:
  compiler:
    - toolchain: gcc
      spec: gcc@11
  unify: true
  mpi:
    spec: cray-mpich@8.1.18.4
    gpu: cuda
  specs:
  - cuda@11.8
  - osu-micro-benchmarks@5.9
  - openblas@0.3.21
  variants:
  - cuda_arch=80
  - +mpi
  - +cuda
prgenv-openacc:
  compiler:
    - toolchain: gcc
      spec: gcc@11
    - toolchain: llvm
      spec: nvhpc
  unify: true
  mpi:
    spec: cray-mpich@8.1.18.4
    gpu: cuda
  specs:
  - osu-micro-benchmarks@5.9%nvhpc
  - cuda@11.8%gcc
  variants:
  - cuda_arch=80
  - +mpi
  - +cuda
```

The software packages are configured as environments, each built using the compiler tool chains built previously, and configured with a (optional) single implementation of Cray-MPICH, that can optionally be configured for CUDA or ROCM support.

- `packages.yaml` and `modules.yaml` make Spack use packages installed on the system and generate modules files, respectively. These follow the YAML specifications for the Spack configuration files with the same names.
- `repo` is an optional path containing a Spack repository[3], for overriding Spack's implementations or providing support for new packages.

Under the hood, the software in the stack is built in a set of Spack environments. For the example stack, there are five environments, illustrated in Fig. 2.

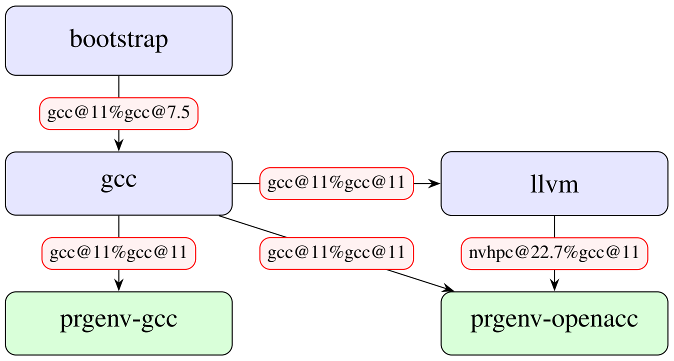


Fig. 2: The dependency graph for the Spack environments that are generated internally by Stackinator to build the example Spack stack. The blue boxes are environments used to build and provide compiler toolchains, and the green boxes show the software environments that are built using the compilers. The red boxes show the compiler toolchain that is used to build the downstream packages.
The bootstrap compiler is built first using the system GCC 7.5, followed by the gcc tool chain, then the llvm tool chain which requires GCC 11. Finally two environments are built: prgenv-gcc with GCC 11, and prgenv-openacc which contains packages built with both GCC 11 and NVHPC 22.7.

The Spack stack requires external packages installed on the base node image, that implement architecture-specific features and support, and can vary between vClusters and over time. For example, the location of libfabric moves every time the installed version of libfabric is upgraded. A set of "cluster configurations" for each of the vClusters on Alps is maintained separately from the recipes in the Stackinator tool:

- A Spack `compilers.yaml` file that specifies the default GCC compiler tool chain on the vCluster (GCC 7.5 at

[3]https://spack.readthedocs.io/en/latest/repositories.html

the time of writing), used to bootstrap the Spack stack build.

- A Spack `packages.yaml` file that specifies externally installed software packages that Spack should never build, including:
  - `libfabric`: the libfabric library installed in `/opt/cray/libfabric/` with CXI provider.
  - `slurm`: vClusters can have different versions of Slurm installed: at the time of writing versions 20.11.9 and 22.05.2 are used.
  - `xpmemm` and `rdma-core`: required by some communication libraries.

B. Stack Configuration

Stackinator follows the familiar configure-build-install workflow used to install software. It provides a CLI tool *stack-config*, that takes a generic recipe that can be built on any (vCluster, mount point) combination, and generates a build path that contains a Makefile, Spack environment descriptions, and a copy of Spack used to build the stack.

If the recipe that describes the example environment in Fig. 2 is in the path `~/recipes/nvidia`, the following *stack-config* command can be used to generate a build configuration:

```
> stack-config -r ~/recipes/nvidia \
            -b /dev/shm/nvidia-build \
            -s hohgant
```

where the generated build path is in `/dev/shm/nvidia-build` and the build is configured for the vCluster Hohgant. A simplified version of the generated build directory structure is illustrated in Fig. 3.

The environment is built using the top-level Makefile, which executes the following steps:

1) (optional) Configure the build cache
2) Call `compilers/Makefile`:
   a) Concretize bootstrap.
   b) Build bootstrap.
   c) Concretize GCC.
   d) Build GCC.
   e) Concretize LLVM (NVHPC).
   f) Build LLVM (NVHPC).
3) Call `environments/Makefile`:
   a) Concretize prgenv-gcc and prgenv-openacc *concurrently*
   b) Build prgenv-gcc and prgenv-openacc*concurrently*
4) Generate `store/config`.
5) (optional) generate `store/modules`.
6) Generate SquashFS image of `store`.

The `compilers.yaml`, `packages.yaml` and `Makefile` for each of the Spack environments are generated in the Makefile using Spack. The commands used generate the respective files required to build the "prgenv-gcc" are summarised:

- `prgenv-gcc/compilers.yaml`:

```
> gcc_prefix= spack -e ../compilers/gcc \
        find --format '{prefix}' gcc@11
> spack compiler find --scope=user \
        $(compiler_bin_dirs $gcc_prefix)
```

```
/dev/shm/nvidia-build
├── Makefile
├── spack
├── compilers
│   ├── Makefile
│   ├── bootstrap
│   │   ├── spack.yaml
│   │   ├── compilers.yaml
│   │   ├── packages.yaml
│   │   └── Makefile
│   ├── gcc
│   │   └── spack.yaml
│   └── llvm
│       └── spack.yaml
├── environments
│   ├── Makefile
│   ├── prgenv-gcc
│   │   └── spack.yaml
│   └── prgenv-openacc
│       └── spack.yaml
└── store
```

Fig. 3: Structure of the build path generated by the Stackinator tool. The files marked in blue are generated by Spack during the build process in each of the five environment paths, and are only shown in the bootstrap path for brevity.

Where the function `compiler_bin_dirs` is a function that returns `bin` path of the compiler, omitted for brevity.

- `prgenv-gcc/packages.yaml`:

```
> spack external find --not-buildable --scope=
   user  perl diffutils gettext
```

- `prgenv-gcc/Makefile`:

```
> spack -e prgenv-gcc/ concretize -f
> spack -e prgenv-gcc/ env depfile \
        -o prgenv-gcc/Makefile
```

First, the environment is concretized (Spack generates the full set of packages and their dependencies for the environment), then a Makefile that facilitates building packages in parallel is generated.

C. Building a Stack

Make is run with an empty environment to improve reproducibility by eliminating the effect of environment variables on the build:

```
> cd /dev/shm/nvidia-build
> env --ignore-environment \
   PATH=/usr/bin:/bin:`pwd`/spack/bin \
   make store.squashfs -j32
```

Two artifacts are generated by the build:

- `store`: a path contains the full software stack, ready to be copied to its final location.
- `store.squashfs`: the path compressed in a SquashFS image, which can be mounted at the mount point.

How CSCS uses SquashFS to provide the user-environments to users is covered in Section III.

The build process uses the Bubblewrap[4] tool when running all spack commands to mount the following locations:

- `/dev/shm/nvidia-build/store` is mounted at the final location for the Spack stack. Thus the stack is built in a location where the user has write permissions, and allows faster in-memory builds (see Section II-E).
- `/dev/null` is mounted at HOME to improve reproducibility by ignoring any Spack configuration in HOME.
- `/dev/shm/nvidia-build/tmp` is mounted at `/tmp` to retain all Spack logs.

D. MPI

Open source MPI distributions – namely OpenMPI, MVAPICH2 and MPICH – are actively developing support for Slingshot 11 with libfabric. However, at the time of writing the only MPI with robust Slingshot 11 support is the Cray-MPICH bundled with the CPE, for which source code is not available.

One of the objectives of this work is to provide software stacks without installing CPE. In order to provide Cray-MPICH, we develop a process for repackaging the compiler wrappers, library, headers and dependencies like PMI that can be installed as a Spack binary package.

Step 1: extract and repackage RPMs

The first step is to create a single directory tree that contains only the required files by extracting them from different RPMs in the CPE distribution downloaded from HPE as illustrated in Fig. 4.

For example, the cray-mpich 8.1.24 packages contains files cherry-picked from the following RPMS in the distribution in CPE 23.3:

1) `cray-mpich-8.1.24-gnu91` for GCC and `cray-mpich-8.1.24-nvidia207` for NVHPC.
2) `cray-mpich-8.1.18-gtl`
   - The GTL (GPU Transport Layer) libraries that implement GPU-aware communication for NVIDIA and AMD GPUs.
3) `cray-pmi-6.1.9`
4) `cray-pmi-devel-6.1.9`

Note that the specific set of RPMs can change from release to release, so the process of extracting the files from RPMs has not been automated. It takes a developer an hour to perform the packaging by hand for each new CPE release.

Separate GCC and NVHPC binary distributions are created because It is not possible to provide a single binary distribution of each cray-mpich version for all compilers, because the Fortran modules in the `include` path are compiler specific.

Step 2: Patch MPI wrappers

As opposed to the CPE, which provides the `CC`, `cc` and `ftn` binary compiler wrappers, the Spack package uses the MPI compiler wrapper scripts `mpicc`, `mpicxx` and `mpifort` that are normally installed in locations like `/opt/cray/pe/mpich/8.1.24/ofi/gnu/9.1/bin`.

```
/
├── bin
│   ├── cray_spawn_server
│   ├── mpic++ -> mpicxx
│   ├── mpicc
│   ├── mpichversion
│   ├── mpicxx
│   ├── mpif77 -> mpifort
│   ├── mpif90 -> mpifort
│   └── mpifort
├── include
│   ├── cray_version.h
│   ├── mpi*.mod
│   ├── mpi*.h
│   ├── pmi*.h
│   ├── pmi2.h
│   └── pmpi_f08.mod
└── lib
    ├── libfmpich.so
    ├── libmpi.[so,a]
    ├── libmpichf90.[so,a]
    ├── libmpich.[so,a]
    ├── libmpifort.[so,a]
    ├── libmpifort_gnu_91.[so,a]
    ├── libmpi_gnu_91.[so,a]
    ├── libmpi_gtl_cuda.[so,a]
    ├── libmpi_gtl_hsa.[so,a]
    ├── libmpl.so
    ├── libopa.so
    ├── libpmi2.[so,a]
    ├── libpmi.[so,a]
    └── libtvmpich.[so,a]
```

Fig. 4: The directory containing cray-mpich and its dependencies. Note that for brevity symlinked library files are removed, and wildcards are used to describe headers.

By default, these wrappers use environment variables set by CPE modules to select the compiler and link architecture-specific libraries. We modify each of the wrappers by parameterizing them on three parameters – `@CC@`, `@@PREFIX@@` and `@@GTL_LIBRARY@@` – that are set by Spack when they are installed:

- hard code the full path to the wrapped compiler;
- set paths `prefix`, `includedir`, to the cray-mpich Spack installation path;
- explicitly link `-lmpi_gtl_cuda` `-lmpi_gtl_hsa` when Cray-MPICH is built with the `+cuda` or `+rocm` variants respectively.
  - this fixes the common runtime error "MPIDI_CRAY_init: GPU_SUPPORT_ENABLED" is requested, but GTL library is not linked.

[4]https://github.com/containers/bubblewrap

Step 3: Create a Spack package

The repackaged Cray-MPICH tar balls are stored in a JFrog Artifactory[5] – a self-hosted artifact store accessible only on the CSCS network that can only be accessed on CSCS systems. A custom Spack package that installs cray-mpich from the tar balls is distributed with Stackinator[6], but can only be run on Alps due to this restriction. To use this process at another site, one would have to create the tar balls, and update the Spack package accordingly.

E. Efficient Stack Builds

Using Spack to build a full software stack – with multiple compilers, libraries and tools – is time and resource consuming. A simple stack based on GCC that provides Python, Cray-MPICH and CUDA, will take in the order of half an hour to build, and environments for the ROCM GPU stack take over two hours to build from scratch on a 64-core Epyc CPU.

It is important to reduce build times where possible, so that maintainers of stacks can iterate and test combinations of packages, and for timely execution of CI/CD pipelines for deploying stacks. This section will document the three strategies that Stackinator uses to achieve this.

Parallelise the build

As illustrated in Fig. 2, building a Spack stack involves building a Spack environments with a DAG of dependencies dictating the order of environment concretization/installation. In turn, installing an environment builds individual packages, which have their own DAG of dependencies.

The Makefiles generated by the Stackinator define the dependencies between environments – facilitating the concurrent concretization and installation of independent environments. The Makefile used to build each environment is generated using the dependency file generation feature of Spack, which can build multiple packages in parallel. A single jobserver is used to parallelise the concurrent environment and package builds.

Perform builds in memory

The Stackinator tool supports building software stacks for installation at mount points where the build process does not have write permissions. For example, on Alps the default mount point is `/user-environment`, which is read-only. The software stack that is to be installed at `/user-environment` is built in the `store` subdirectory of the build path, where the builder has write permissions. The built process uses Bubblewrap to mount the `store` path at `/user-environment` for all calls to Spack. In this way, all software is built "in place" and no relocation is required.

Build times can be signficantly reduced by creating the build path in memory, for example in `/dev/shm/build`, so that all of the dependencies are built and stored in memory, instead of on a slower shared file system. All of the Cray EX nodes on Alps have 512 GB of memory, which is sufficient for building software stacks, though it is important that the memory is cleaned up, preferably via an automated policy. See Section IV-A for a comparison between building in memory and on a shared filesystem.

Cache previously built packages

The most effective way to reduce build times is to not rebuild previously built packages. Spack provides build caches, which facilitate pushing and pulling pre-built packages to S3 buckets or a file system. Stackinator recipes can include optional `mirrors.yaml` file and a private key, to enable build caches. Providing the location of a build cache with `mirrors.yaml` will enable pulling packages from the cache, and if a key is provided Stackinator will also push all packages to the build cache as they are built.

III. Deployment

A. SquashFS artifacts

Software stacks can be deployed by copying them to a path on a shared file system, for example if the site-policy is to install `/apps/stacks/<env-name>`, the process for building a climate software stack `climate-23.3` would be:

1) Configure the build with `stack-config` with mount point `/apps/stacks/climate-23.3`, and build in `/dev/shm/build/climate-23.3`, to reduce the build time compared to building in place on the shared file system.
2) Once built, copy `/dev/shm/build/climate-23.3/store/*` to `/apps/stacks/climate-23.3`.

Installing software stacks in shared file systems has some downsides, namely:

- The user-experience is affected by file system performance – configuration and compilation access many small files which are not well-suited to GPFS and LUSTRE.
- High storage overheads – for example, a software stack with CUDA and NVHPC SDK requires at least 30 GB uncompressed.
- Upgrading the version of a stack by installing it in a new path requires changing that path in all downstream user scripts and workflows.
- Users will combine software from different stacks, often by accident, leading to difficult to debug linking and runtime bugs.

To address these issues, CSCS deploys the software stacks as compressed SquashFS images of the directory containing the software, Spack configuration, modules and meta-data. Squashfs is an efficient and compressed read-only file system that offers is well-suited for distributing software stacks, for the following reasons:

- SquashFS supports both compression and deduplication, resulting in significantly reduced storage requirements. The software stack for Meteo Swiss requires 34 GB uncompressed, and 13 GB as a compressed SquashFS image.
- Each stack is a single compressed file that includes an entire software stack, making it easy to manage in DevOps pipelines and archives:

[5] https://jfrog.com/artifactory/
[6] See `stackinator/repos` in the repository https://github.com/eth-cscs/stackinator.

  - each CI/CD pipeline generates a single binary artifact;
  - programming environments can be versioned and archived as binary artifacts;
  - new stacks can be tested transparently by mounting a pre-release stack at a common mount point.
- Users can mount stacks on command using command line utilities or Slurm plugins. Only one stack is mounted at a time – reducing confusion about which software is being used in a workflow. Multiple users on the same login or compute nodes can mount different software stacks without side-effects on other user's environments.
- Consistent and reproducable performance for workloads that access many small files by virtue of the whole stack being stored in a single file and file-system caching.

B. CLI Utilities

Non-privileged users are able to mount SquashFS images at runtime using the `squashfs-mount` command line utility, which is a small `setuid` executable that creates a new mount namespace, mounts the SquashFS file through `libmount`, drops privileges and executes a given command. This procedure is very similar to SquashFS-based HPC container runtimes such as Apptainer and Sarus.

For example,

```
squashfs-mount image.squashfs /user-environment bash
```

starts a bash shell in which `image.squashfs` is mounted at `/user-environment`. Thanks to mount namespaces, the mount is not visible to other processes or users.

The utility is open source, available on GitHub and includes RPMs for installation on Cray EX.

C. Slurm Integration

For a software stack to be available when a job runs, the stack must be mounted in the namespace of the process executing the submission script and any commands launched on other compute nodes. To accomplish this we developed a Slurm plugin that mounts a software stack based upon the same namespace mechanism used by the command line utilities discussed earlier. As the mount point is not globally visible, nodes can run multiple jobs with different stacks, either from the same user or different users. Crucially, only one image is mounted per node not per parallel task. Clean up of the mount point and software stack is performed automatically once the parent process terminates. The plugin is publicly available on GitHub[7].

The plugin is designed to be as transparent to a user's workflow as possible. This reduces work for users and system administrators, as the plugin and SquashFS-based software stack concept can easily be integrated into existing systems and workflows. In particular, the plugin works with `squashfs-mount` to detect which software stacks (and their mount points) are active. These are taken as default values to the plugin which will make them available on the compute nodes. A different stack can be specified as a command-line option to any of the Slurm submission commands (e.g., sbatch or srun). This flexibility ensures consistency between the login environment and the execution environment, while also allowing the user to use different stacks *within* a script.

The plugin is written in C++ using the SPANK API for Slurm plugins. Root-level access is required to mount the SquashFS image, which means the namespace creation and mounting code is located in the API function that Slurm runs in privileged mode. Installation also requires system-administrator privileges. Typically, this means adding the path of the plugin library directly to the Slurm plugin configuration file `/etc/slurm/plugstack.conf` or in a plugin-specific configuration file under `/etc/slurm/plugstack.conf.d/`.

In the following example the user is on the login node, mounts a user environment using `squashfs-mount`, and starts a `bash` shell. This environment provides `gcc` under `/user-environments/bin`. The user can access this version of `gcc` when running on a compute node via `srun` because the Slurm plugin will recognize the mounted user environment and provide it on the compute node as well.

```
$ squashfs-mount compilers.sqfs /user-environment
    bash
$ srun /user-environment/bin/gcc test.c
```

The next example demonstrates using different environments throughout a Slurm submission script. When the script executes the plugin will have already mounted the user environment (debuggers.sqfs) mounted at the time the job was submitted. The first command runs a bash script with a different user environment (compilers.sqfs) mounted to compile a test program. This test program is run through the debugger found in the original user environment. The subsequent commands show an alternative procedure where the compiler environment is specified directly to srun to compile the test program. Then the debugger is again run, this time via srun.

```
$ squashfs-mount debuggers.sqfs /user-environment
    bash
$ sbatch test.job
$ cat test.job
#!/bin/bash
#SBATCH -J test_job
#SBATCH -t 10:00:00

squashfs-mount compilers.sqfs /user-environment bash
     <<EOF
srun /user-environments/bin/gcc test.c
EOF

/user-environment/bin/gdb -ex run a.out

srun --uenv-file=compilers.sqfs /user-environments/
    bin/gcc test.c

srun /user-environment/bin/gdb -ex run a.out
```

D. CI/CD

A CI/CD pipeline is under development for the Spack-stacks provided by CSCS to the users of vClusters on Alps. Each software stack is maintained as a Stackinator recipe in a public

[7]https://github.com/eth-cscs/slurm-uenv-mount

GitHub repository[8]. The recipes are publicly available, so that user-groups can use them as the basis for building their own software stacks.

CSCS has a CI/CD service, that can be configured to respond to web hooks from a GitHub repository[9]. When an authorized user requests that the pipeline for a recipe be run from pull request (PR) on the GitHub recipe repository, a webhook alerts the CSCS CI/CD service. Each CI/CD job provides a tuple (Node-type, Cluster), where node-type is currently one of A100, Mi200, Zen2 or Zen3. The service launches a build task on the target node type, using the configuration for the target cluster (that configures Spack to use the system compiler, libfabric, Slurm, etc).

The *build stage* performs the following steps:

- Download and install Stackinator.
- Configure the recipe to use a common build-cache for all CI/CD pipelines.
- Run Stackinator `stack-config` to configure a build path in `/dev/shm`.
- Build the stack in parallel.
- Push the squashfs image to a job-specific path in the CSCS JFrog[10] artifactory.

Development of the CI/CD pipeline for images is on-going, with the following stages under development for release in Q3 2023:

- *test stage*: ReFrame[11] [6] inspects the uenv metadata listing the features offered by each programming environment. It then launches a list of checks to ensure that the provided features are working correctly and the performance is optimal.
- *deploy stage*: deploy the image to a production artifactory where it can be accessed by users with a simple CLI interface.

IV. Results and performance

A. Efficient Deployment

This benchmark tests the impact of optimisations to reduce build time of spack stacks described in Section II-E, namely building in memory and caching previously-built packages.

For this demonstration, we built a software stack that has all of the dependencies required to develop Arbor [1], [2], a Neuroscience application written in C++ and Python. Arbor has a extensive list of dependencies, including C++ libraries, Python and Python packages.

We time the time taken to run make on a clean build, which includes the time taken to bootstrap Spack, concretise and build all of the packages and generate the compressed SquashFS image in four different scenarios:

- **scratch**: Build on an HPE Cray ClusterStor E1000 Scratch file system.
- **memory**: Build in `/dev/shm`, i.e. *in memory*.
- **cache**: Build in `/dev/shm` using a Spack build cache that has all of the packages
- **partial**: Build in `/dev/shm` using a Spack build cache where the version of Python in the recipe is changed to a version that is not in the cache.

Scenario 2 quantifies the effect of building in memory, and scenarios 3 and 4 illustrate the additional benefits of using build caches.

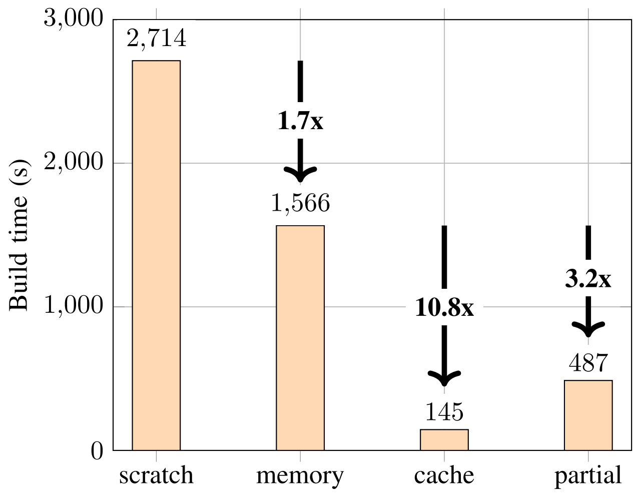


Fig. 5: Reduction in time to build a complete Spack stack when building in memory and using Spack build caches compared with building on the scratch filesystem.

Fig. 5 shows that building the image on Scratch takes 45 minutes, which is reduced to 26 minutes when building in memory – a significant 1.7× reduction in build time. Less than 3 minutes are required when all packages are available in a build cache, and less than 10 minutes to build the full stack when the version of Python in the recipe changed, which required rebuilding over 30 packages, including Python, py-numpy and py-mpi4py, which are non-trivial to build.

The *partial* reflects the most common scenario, because the typical CI/CD and image development process requires rebuilding an image with small changes, so that only some of the packages need to be rebuilt between runs. Furthermore, the bootstrap and compiler toolchains are typically identical between different images – e.g., once GCC 11.3.0 has been built for one stack, it can be reused without change in another.

B. Developer Productivity

We now test whether using compilers and libraries installed via SquashFS has any impact on the time taken to configure applications on the command line, which has a direct impact on developer productivity. In these tests we will use the Arbor programming environment used in the previous tests – where the environment is installed in three different ways:

1) **squashfs**: a SquashFS image mounted at `/user-environment`.
2) **scratch**: installed on the Scratch file system.

[8]https://github.com/eth-cscs/alps-spack-stacks
[9]https://gitlab.com/cscs-ci/ci-testing/containerised_ci_doc
[10]https://jfrog.com/artifactory
[11]https://github.com/reframe-hpc/reframe

3) **memory**: stored in `/dev/shm` and mounted at `/user-environment` with Bubblewrap.

First, we look at the time required to compile a single "hello world" C and C++ files using the GNU compiler provided by the stack:
`hello.c`:

```
#include <stdio.h>
int main(void) {
    printf("hello world\n");
    return 0;
}
```

`hello.cpp`:

```
#include <iostream>
int main(void) {
    std::cout << "hello world\n";
    return 0;
}
```

As illustrated in Table I, the compilation times are within 1% when the compiler toolchain is in memory or mounted via SquashFS, and between 4-11% slower when the toolchain is installed on the Scratch filesystem.

	squashfs	scratch	memory
C	31.1	34.7 (+11%)	31.2 (< 1%)
C++	266	276 (+3.8%)	264 (< 1%)

TABLE I: The time taken (in ms) to compile simple hello world C and C++ files using the programming stack installed in different locations.

A more involved example is to build Arbor using the stack developed above. This is broken into two steps:

1) **configure**: run CMake to configure a build with MPI and Python enabled, and use generated build files for Ninja: `CC=mpicc CXX=mpic++ cmake ../arbor -DARB_WITH_MPI=on -DARB_WITH_PYTHON=on -G Ninja`.
2) **build**: run Ninja to build Arbor.

To isolate the file system overheads of accessing the stack, the Arbor source code and build path are in `/dev/shm`. The results in Table II show that the SquashFS mount and in memory are equivalent, which there is a performance penalty of between 8-23% on Scratch.

	squashfs	scratch	memory
configure	2.52	3.09 (+23%)	2.53 (< 1%)
build	33.9	36.7 (+8%)	33.8 (< 1%)

TABLE II: The time taken (in s) to configure and build Arbor.

We note the tests in this section were run when the file system was not in heavy use, and smaller differences were observed when tested on a flash-based Lustre store – so while significant, the performance benefits of using SquashFS over LUSTRE reported here might not justify using SquashFS. However, in our experience performance of workloads that access many files in a SquashFS stack is very consistent, regardless of load on the system, and when the SquashFS image itself is stored in Scratch. On the other hand, compilation and configuration times vary greatly for software stacks installed on Lustre or GPFS filesystems – when the file system is under heavy load compilation can be much slower. As such, SquashFS is both faster and more consistent and predictable than installing software on shared file systems, improving the quality of the user experience on our systems.

C. *Benchmarks and Applications*

In this section we present benchmarks compiled with CPE and Spack stacks on the same system, everything else being equal. The purpose of the benchmarks is not to compare the performance of node types, or evaluate the efficiency of the benchmarks, instead the objective is to demonstrate equivalent performance of the benchmarks when built using the CPE software stack and spack-stacks. All of the benchmarks use Cray-MPICH, in order to understand whether repackaging Cray-MPICH for installation with Spack has any impact on performance.

MicroBenchmark: OSU

Selected OSU microbenchmarks[12] were run on the vCluster Clariden, which has nodes with 4 NVIDIA A100 GPUs, a single socket AMD Zen3 Milan CPU, and 4 Slingshot 11 NICs. The following three selected benchmarks run in both host-host and device-device configurations:

- **osu_bw**: Point to Point bandwidth test. The benchmark was run between two ranks on different nodes.
- **osu_latency**: Point to Point latency test. The benchmark was run between two ranks on different nodes.
- **osu_alltoall**: All to all latency test. The benchmark was run between 16 ranks on 4 nodes, with one GPU per rank when running device-device tests.

The following wrapper script was used to launch all of the jobs (note that the GPU flags will have no impact on the CPU-only runs).

```
export LOCAL_RANK=$SLURM_LOCALID
export GLOBAL_RANK=$SLURM_PROCID

export GPUS=(3 2 1 0)
export NUMA_NODE=$LOCAL_RANK
export CUDA_VISIBLE_DEVICES=${GPUS[$NUMA_NODE]}

export MPICH_GPU_SUPPORT_ENABLED=1

numactl --cpunodebind=$NUMA_NODE \
  --membind=$NUMA_NODE \
  $exe
```

The wrapper script ensures optimal affinity of GPUs with NUMA regions, and we let Cray-MPICH select the NIC in each case – when using both CPE and Spack stacks the same NIC was assigned.

The versions of compilers and tools did not match exactly:

	CPE	Spack Stack
osu-benchmark	5.9	5.9
cray-mpich	8.1.21	8.1.24
gcc	11.2	11.3
cuda	11.6	11.8

[12]https://mvapich.cse.ohio-state.edu/benchmarks/

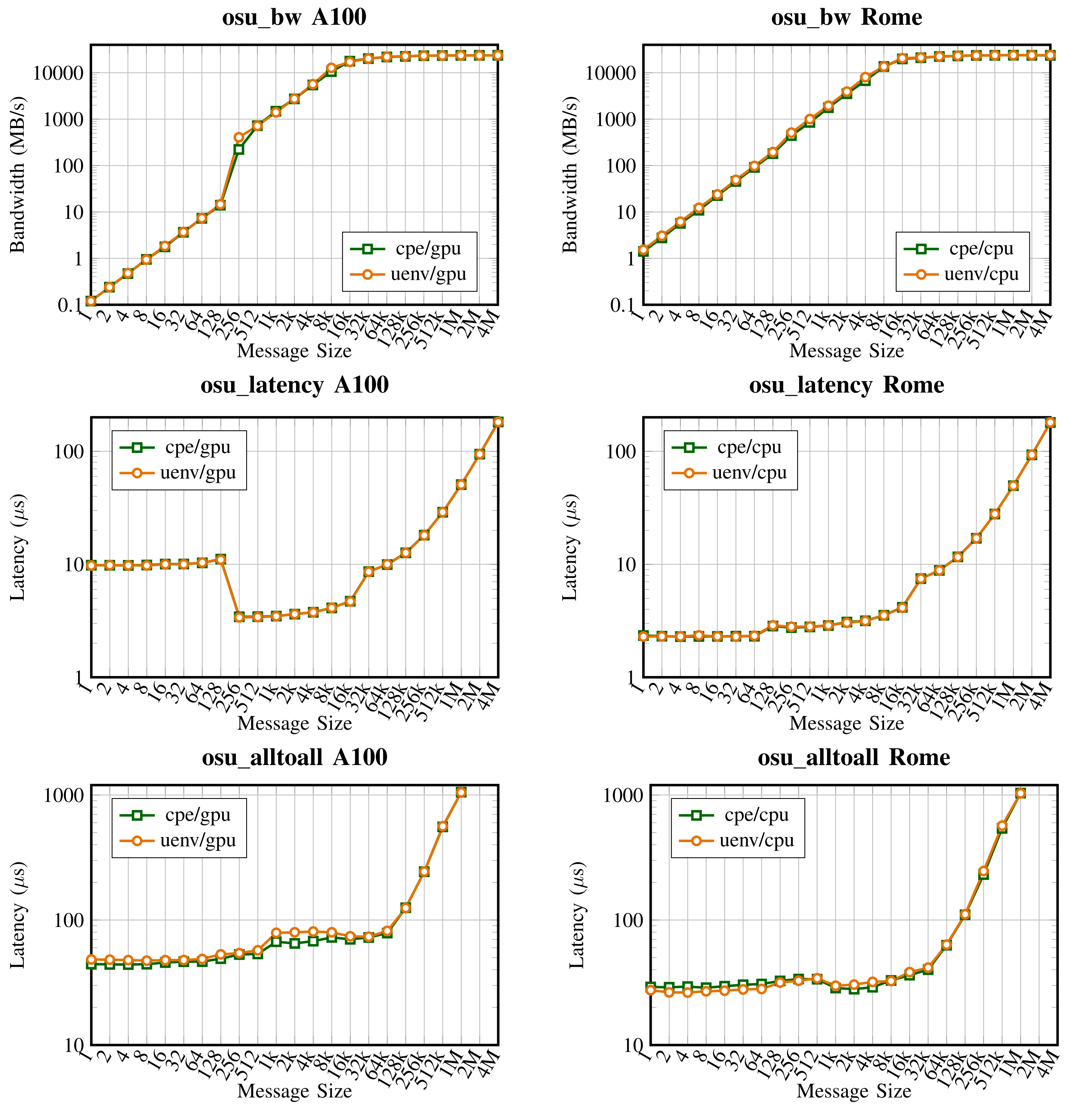


Fig. 6: OSU benchmark results comparing cray-mpich performance when built using CPE and spack-stacks (uenv) for host-host (cpu) and device-device (gpu) configurations.

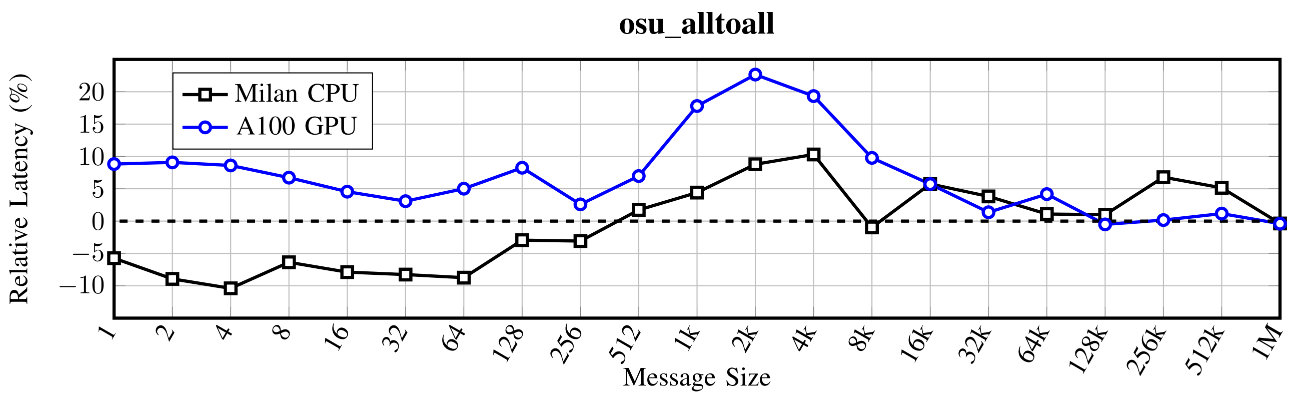


Fig. 7: The relative latency for the osu_all2all benchmark between cray-mpich in a Spack stacks and in CPE, where zero means equal latency, above zero means Spack stack has higher latency and below zero CPE has higher latency. Results are shown for host-host (Milan) and device-device (A100)

The most recent version of CPE installed on the system was v22.12, and the stack was built using the version of cray-mpich in v23.3. However, earlier benchmarks and tests using other versions of cray-mpich from CPE and Spack stacks gave the same results.

The OSU benchmark results, plotted in Fig. 6, illustrate that there is no discernable benefit either way of using cray-mpich from CPE or installed via Spack for the point-to-point bandwidth and latency results. For the all-to-all collective there were more significant relative differences for messages under 32K in size, as illustrated in Fig. 7:

- messages (≤ 256 bytes): CPE had higher latency for host-memory, and Spack stacks had consistently higher latency on the GPU.
- messages (512–16k bytes): Spack stacks had between 5-20% higher latency.

More testing where we pin down the exact same version of cray-mpich and other dependencies would be required to understand whether the difference is caused by how cray-mpich is installed in Spack stacks.

Application Benchmark: GROMACS

A strong-scaling GROMACS benchmark was performed using the same version of GROMACS built using the CPE and a Spack-stack. GROMACS[13] is free and open-source software suite for high-performance molecular dynamics and output analysis. The GROMACS version used is 2021.5 in single precision, while the versions of compiler and cray-mpich did not match exactly between the CPE and spack-stack:

	CPE	Spack Stack
gromacs	2021.5	2021.5
fftw	3.3.10	3.3.10
openblas	0.3.21	0.3.21
cray-mpich	8.1.21	8.1.24
gcc	11.2	11.3

[13]https://www.gromacs.org/

Note that when using CPE FFTW and OpenBLAS were built using Spack instead of using the cray-fftw and cray-libsci modules from CPE in order to isolate the impact of cray-mpich.

The simulations were run on nodes with a single socket AMD Zen3 Milan CPU, and 4 Slingshot 11 NICs. For the strong scaling benchmarks, a 1.4-million atom system (a pair of hEGFR Dimers of 1IVO and 1NQL) is used, included in the HECBioSim benchmarks suite[14]. The number of MPI tasks per node was kept constant at 64 with one OpenMP thread per rank, and the number of nodes was scaled from 1 to 12.

The benchmark strong scales well, as shown in Fig. 8, with a difference of 1%-1.5% between the CPE and the Spack-stack with no clear advantage between the two.

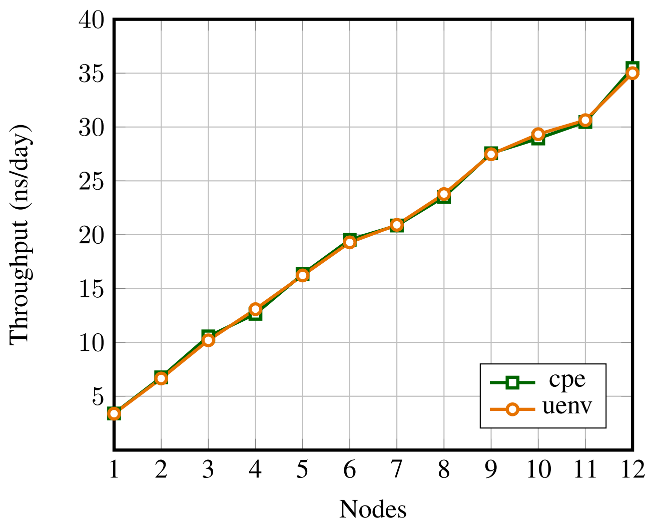


Fig. 8: GROMACS strong scaling measured in ns/day (higher is better) when built using spack stacks and CPE.

[14]https://www.hecbiosim.ac.uk/access-hpc/benchmarks

Application Benchmark: SPH-EXA

The SPH-EXA[15] project is a multidisciplinary effort that looks to scale the Smoothed Particle Hydrodynamics (SPH) method to enable exascale hydrodynamics simulations for the fields of Cosmology and Astrophysics. This section focuses on comparing the behavior of the code built with and without the Stackinator tool. For this, we built two Stackinator stacks: one for CUDA and one for ROCM. The compiler and libraries included in these images were based on cray-mpich/8.1.21, in addition to gcc/11.x and cuda/11.8 or hip/5.2 for the CUDA and ROCM stacks respectively. Next, the stacks were mounted to access the compiler and libraries for building our MPI+OpenMP+CUDA and MPI+OpenMP+HIP versions of the code. Finally, we executed the executables and compared the results with those of the same code built with the Cray Programming Environment (CPE). The primary focus in this context is the runtime behavior of the code rather than the performance of the code itself.

Fig. 9 shows the performance obtained for the codes executing the Sedov–Taylor[16] blast wave explosion test case with 400^3 particles per gpu and 40 time-steps, and for the MPI+OpenMP CPU-only version of the code with 483^3 particles per compute node.

The results show that the squashfs-based executables deliver competitive performance with that of the CPE based executables on both GPU architectures and multicore: the Spack-stack was between 4-10% faster on A100 GPUs, between 0-4% slower on AMD GPU and the CPU results are within 2%.

D. Analysis Tools

1) Debugging tools: This section focuses on showing the usage of the Forge DDT debugger in a user-environment.

Forge DDT can be launched using Express Launch (`ddt srun myexe`), Reverse Connect (`ddt --connect srun myexe`) or Offline mode (`ddt --offline srun myexe`). Express Launch requires a good X11 connection hence it is recommended to use the Reverse Connect method, which uses the Remote Client to debug remote jobs while running the user interface on a local machine. In addition, Forge DDT can be configured to integrate with queuing systems, allowing for the submission of jobs directly from the user interface. The debugger can also open and debug one or more core files after an application terminates or attach to running programs.

In a classic workflow, users build a debug version of their code with the -g and -G compilation flags and submit a job with the sbatch slurm command. For example, the `native.slurm` script:

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=16
#SBATCH --threads-per-core=1
#SBATCH --output=job.out
#SBATCH --error=job.err
#SBATCH --time=0:15:0
```

[15]https://github.com/unibas-dmi-hpc/SPH-EXA
[16]https://doi.org/10.48550/arxiv.2202.02840

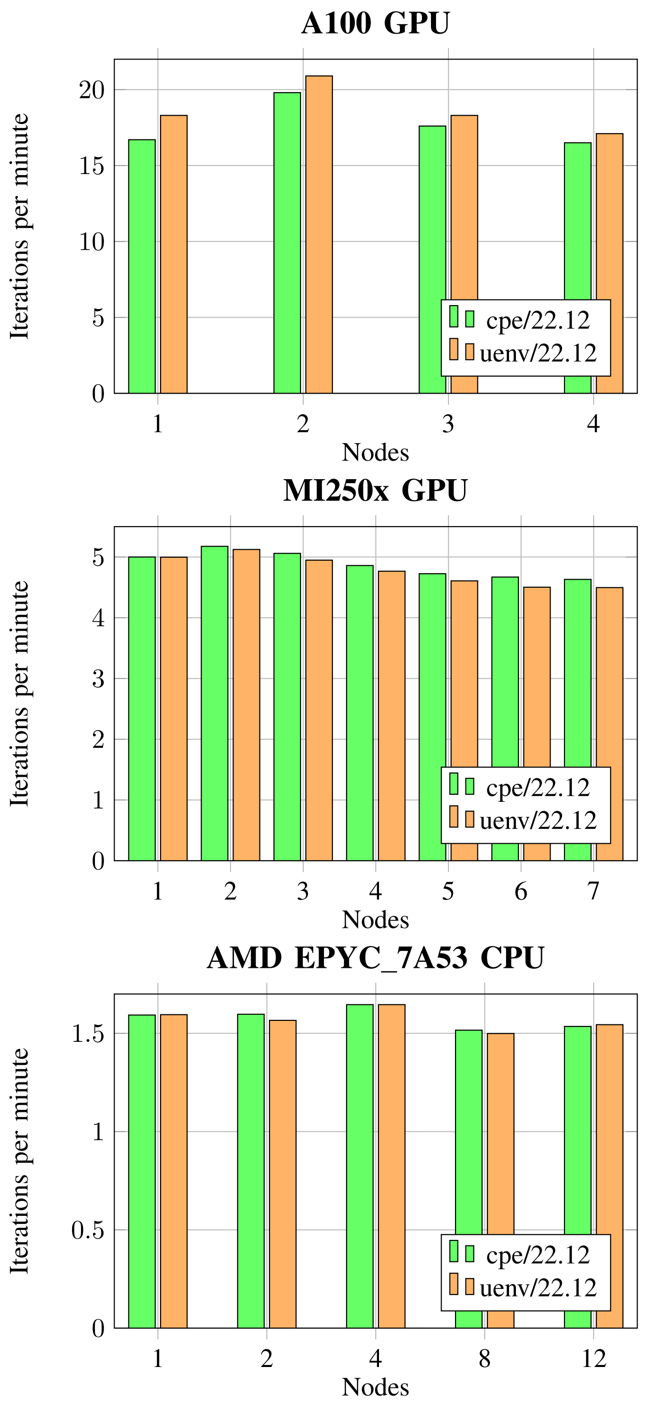
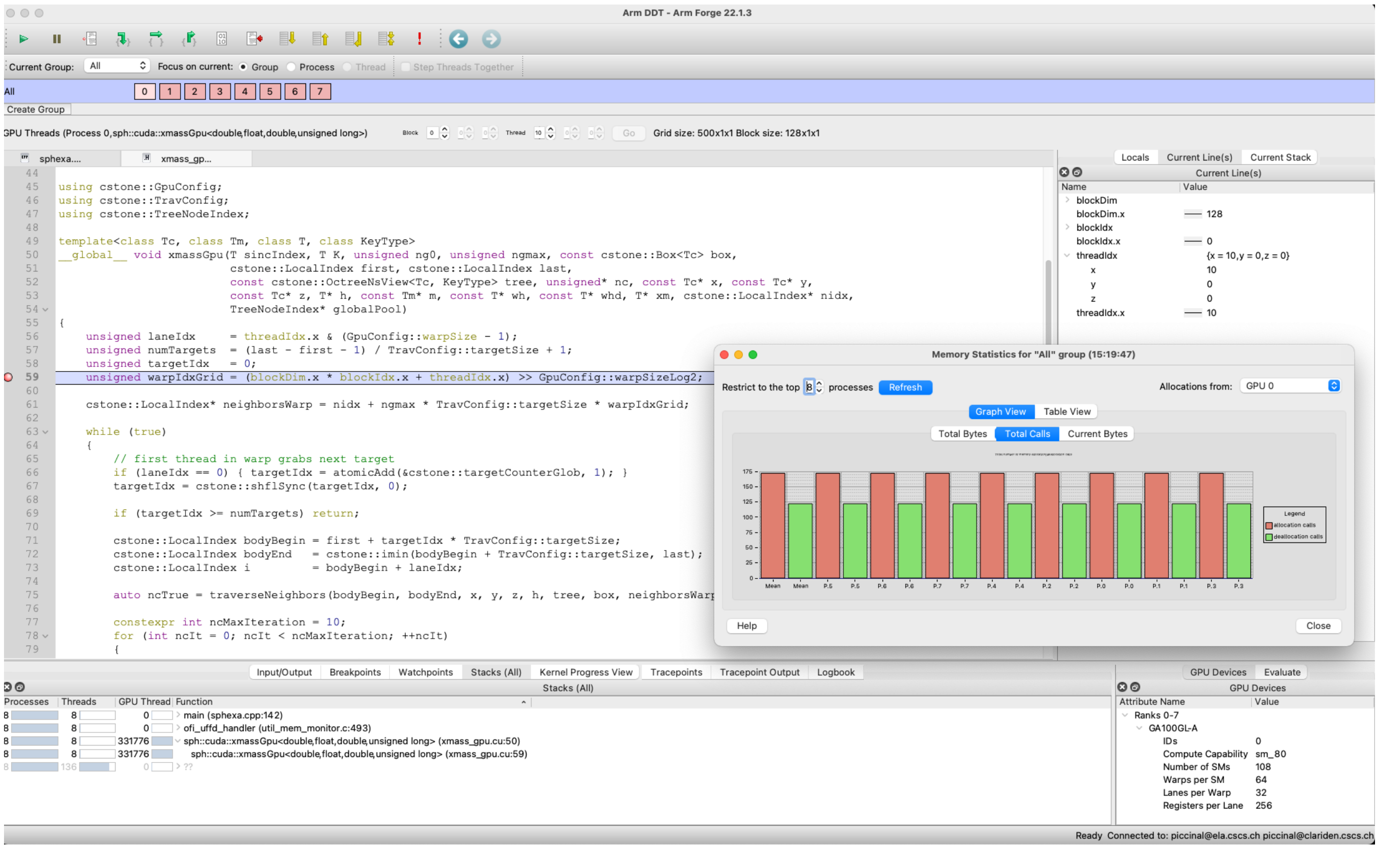


Fig. 9: Weak scaling results on A100 and Mi250x GPU and AMD CPU nodes for SPH-EXA (higher is better).



```
using cstone::GpuConfig;
using cstone::TravConfig;
using cstone::TreeNodeIndex;

template<class Tc, class Tm, class T, class KeyType>
__global__ void xmassGpu(T sincIndex, T K, unsigned ng0, unsigned ngmax, const cstone::Box<Tc> box,
                         cstone::LocalIndex first, cstone::LocalIndex last,
                         const cstone::OctreeNsView<Tc, KeyType> tree, unsigned* nc, const Tc* x, const Tc* y,
                         const Tc* z, T* h, const Tm* m, const T* wh, const T* whd, T* xm, cstone::LocalIndex* nidx,
                         TreeNodeIndex* globalPool)
{
    unsigned laneIdx     = threadIdx.x & (GpuConfig::warpSize - 1);
    unsigned numTargets  = (last - first - 1) / TravConfig::targetSize + 1;
    unsigned targetIdx   = 0;
    unsigned warpIdxGrid = (blockDim.x * blockIdx.x + threadIdx.x) >> GpuConfig::warpSizeLog2;

    cstone::LocalIndex* neighborsWarp = nidx + ngmax * TravConfig::targetSize * warpIdxGrid;

    while (true)
    {
        // first thread in warp grabs next target
        if (laneIdx == 0) { targetIdx = atomicAdd(&cstone::targetCounterGlob, 1); }
        targetIdx = cstone::shflSync(targetIdx, 0);

        if (targetIdx >= numTargets) return;

        cstone::LocalIndex bodyBegin = first + targetIdx * TravConfig::targetSize;
        cstone::LocalIndex bodyEnd   = cstone::imin(bodyBegin + TravConfig::targetSize, last);
        cstone::LocalIndex i         = bodyBegin + laneIdx;

        auto ncTrue = traverseNeighbors(bodyBegin, bodyEnd, x, y, z, h, tree, box, neighborsWarp

        constexpr int ncMaxIteration = 10;
        for (int ncIt = 0; ncIt < ncMaxIteration; ++ncIt)
        {
```

GPU Threads (Process 0,sph::cuda::xmassGpu<double,float,double,unsigned long>) Grid size: 500x1x1 Block size: 128x1x1

Name	Value
blockDim	
blockDim.x	128
blockIdx	
blockIdx.x	0
threadIdx	{x = 10,y = 0,z = 0}
x	10
y	0
z	0
threadIdx.x	10

Memory Statistics for "All" group (15:19:47)

Processes	Threads	GPU Thread	Function
8	8	0	main (sphexa.cpp:142)
8	8	0	ofi_uffd_handler (util_mem_monitor.c:493)
8	8	331776	sph::cuda::xmassGpu<double,float,double,unsigned long> (xmass_gpu.cu:50)
8	8	331776	sph::cuda::xmassGpu<double,float,double,unsigned long> (xmass_gpu.cu:59)
8	136	0	??

Attribute Name	Value
Ranks 0-7	
GA100GL-A	
IDs	0
Compute Capability	sm_80
Number of SMs	108
Warps per SM	64
Lanes per Warp	32
Registers per Lane	256

Ready Connected to: piccinal@ela.cscs.ch piccinal@clariden.cscs.ch

Arm DDT - Arm Forge 22.1.3
Current Group: All
Focus on current: Group Process Thread
Step Threads Together
All 0 1 2 3 4 5 6 7
Create Group
Block 0 Thread 10 Go
sphexa.... xmass_gp...
Locals Current Line(s) Current Stack
Restrict to the top 8 processes Refresh
Allocations from: GPU 0
Graph View Table View
Total Bytes Total Calls Current Bytes
Legend
allocation calls
deallocation calls
Help Close
Input/Output Breakpoints Watchpoints Stacks (All) Kernel Progress View Tracepoints Tracepoint Output Logbook
GPU Devices Evaluate

Fig. 10: Debugging SPH-EXA with Forge DDT.

```
#SBATCH --partition=nvgpu
#
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
module load cray/22.11
module swap PrgEnv-cray PrgEnv-gnu
module swap gcc/11.2.0 CUDAcore/11.8.0
#
ddt --connect \
srun --cpus-per-task=16  \
--cpu-bind=verbose,none ./cuda_visible_devices.sh \
./native.exe
```

will load the Cray Programming Environment (CPE) and launch the debugger with the `ddt --connect` command. As the job starts, the users can use the remote client installed on their local machine to debug the code.

When running in a user-environment, it is required to manually launch the debugger. For example, the `uenv.slurm` script:

```
#!/bin/bash
#SBATCH --ntasks=8
#SBATCH --ntasks-per-node=4
#SBATCH --cpus-per-task=16
#SBATCH --threads-per-core=1
#SBATCH --output=job.out
#SBATCH --error=job.err
#SBATCH --time=0:15:0
#SBATCH --partition=nvgpu
#SBATCH --uenv-file=forge2213.squashfs
#SBATCH --uenv-mount=/user-environment
#
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
# export ALLINEA_SET_SYSROOT="/"
module use /user-environment/modules
module load arm-forge cray-mpich cuda gcc
#
srun --cpus-per-task=16  \
--cpu-bind=verbose,none ./cuda_visible_devices.sh \
forge-client ./uenv.exe
```

will mount the uenv-file, load the User Environment and launch the debugger with the `forge-client` command. Similarly to the previous case, users can debug the code when the job starts. Fig. 10 shows the Forge DDT remote client connected to a job running the SPH-EXA code. DDT can be used to debug parallel programs in both scenarios. Attaching to running programs with Forge DDT in a user environment requires additional testing as the process running the application and the Forge DDT process are not necessarily in the same namespace.

A100	H2D	D2H	D2D
4	15,782	8,636	16,819
8	18,632	6,593	16,845
12	30,834	8,808	16,879
16	30,849	7,201	16,898

TABLE III: CUDA memcpy

2) *Performance tools:* Table III shows the amount of CUDA memory copies (in MB) for the Sedov–Taylor test case.

The performance data was collected with the NVIDIA Nsight Systems[17] tool. The transfer sizes (in MB) for Host to Device (H2D), Device to Host (D2H) and Device to Device (D2D) for simulations with uenv and without uenv (cpe only) are equal, demonstrating that the performance tool can be used in both scenarios.

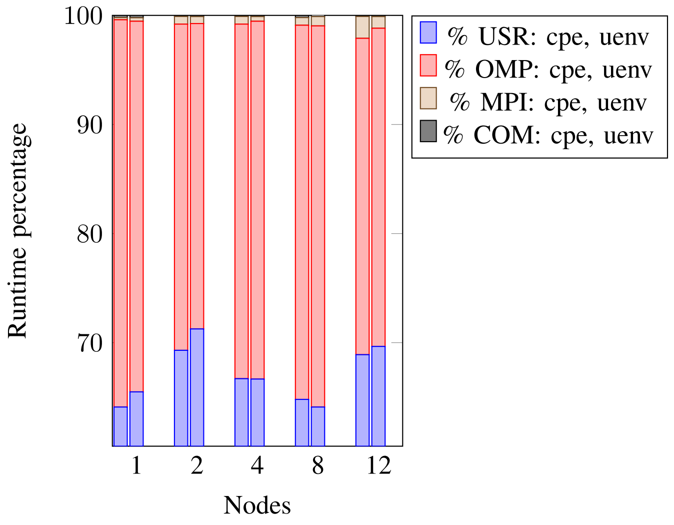


Fig. 11: Weak scaling profiling results on AMD EPYC 7A53 CPU nodes for SPH-EXA.

Fig. 11 shows profiling results for the Sedov–Taylor test case on CPU nodes. The performance data was collected with the Score-P[18] tool (version 8.1). The breakdown of the runtime into different regions such as USER, OpenMP and MPI demonstrates that the performance tool can be used in both environments.

V. Future work

The user-environmnents on Cray EX systems presented here is currently being used internally by CSCS software development teams, and by the MeteoSwiss in preparation for their next operational weather forecast system. The service will be rolled out to external users of vCluster users over 2023, and there will be further development to extend features and provide a robust service.

The CI/CD pipeline is a high priority, with integration with the ReFrame testing framework to ensure that software stacks are corrent and performant.

We plan to provide a mechanism for the command line tools and Slurm plugin to mount multiple images – for which the main motivating use case is providing debugger and profiler toolchains alongside programming stacks.

A command line tool that provides a singled interface for users to query, download and interact with environments is under development.

We will discuss collaboration with HPE to provide Cray-MPICH and other CPE software packages via Spack stacks, as well as extending our current experimental support for other MPI distributions on Cray EX.

Finally, while the tools are designed by CSCS and for use on Alps, we would welcome collaboration with other sites.

References

[1] N. Abi Akar, B. Cumming, V. Karakasis, A. Küsters, W. Klijn, A. Peyser, and S. Yates. Arbor — A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 274–282, feb 2019.

[2] N. A. Akar, J. Biddiscombe, B. Cumming, M. Kabic, V. Karakasis, W. Klijn, A. Küsters, A. Peyser, S. Yates, T. Hater, B. Huisman, E. Hagen, R. D. Schepper, C. Linssen, H. Stoppels, S. Schmitt, F. Huber, M. Engelen, F. Bösch, J. Luboeinski, S. Frasch, L. Drescher, and L. Landsmeer. Arbor library v0.8.1, Nov. 2022.

[3] S. R. Alam, M. Gila, M. Klein, M. Martinasso, and T. C. Schulthess. Versatile software-defined hpc and cloud clusters on alps supercomputer for diverse workflows. *The International Journal of High Performance Computing Applications*, 0(0):10943420231167811, 0.

[4] P. Forai, K. Hoste, G. Peretti-Pezzi, and B. Bode. Making scientific software installation reproducible on cray systems using easybuild. In *Proceedings of the Cray Users Group Meeting (CUG2016)*, 2016.

[5] T. Gamblin, M. P. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and W. S. Futral. The Spack Package Manager: Bringing order to HPC software chaos. In *Supercomputing 2015 (SC'15)*, Austin, Texas, November 15-20 2015.

[6] T. Manitaras, V. Karakasis, J. Otero, E. Koutsaniti, J.-G. Piccinali, R. Sarmiento, C. Bignamini, V. H. Rusu, A. Jocksch, M. Kraushaar, L. Marsella, S. Keller, S. Omlin, S. Kliavinek, H. Mendonça, M. Giordano, M. Turner, G. Lo-Re, M. Boissonneault, S. Leak, M. Paipuri, V. Sochat, J. Favre, S. Moors, Z.-Q. You, Åke Sandgren, J. Morrison, X. Lapillonne, and E. Birngruber. reframe-hpc/reframe: Reframe 4.2, Apr. 2023.

[17] https://developer.nvidia.com/nsight-systems
[18] https://score-p.org