CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Deploying Alternative User Environments on Alps

CUG23 – Helsinki

B. Cumming, J. Coles, T-I. Manitaras, J-G. Piccinali, S. Pintarelli, H. Stoppels

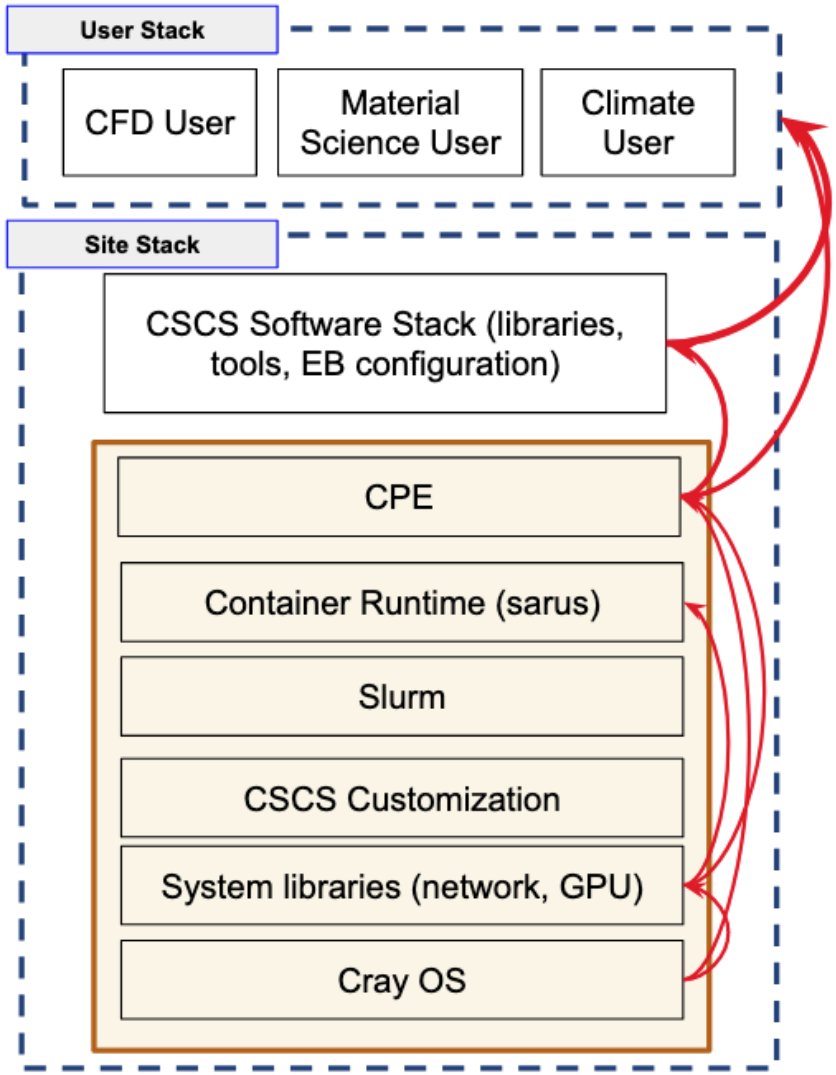
May 10, 2023

Alps is the new HPE Cray EX-based infrastructure at CSCS.

Consolidate separate service-specific clusters onto a single infrastructure – versatile software-defined clusters (vClusters) with workload-specific software environment, scheduler, storage and network isolation.

. . . software stack deployment won't scale with our existing deployment model. . .

Monolithic Software Stacks



Sites provide CPE – then provide software built on top:

- install all the software for all the users on a shared file system;
- use CPE modules + site modules for environment customisation.

CPE presents challenges as a software stack foundation:

- changes every 3 months
- has a large surface area of possible bugs and regressions.
- deploying fixes takes minimum 3-6 months.

Bespoke SW stacks

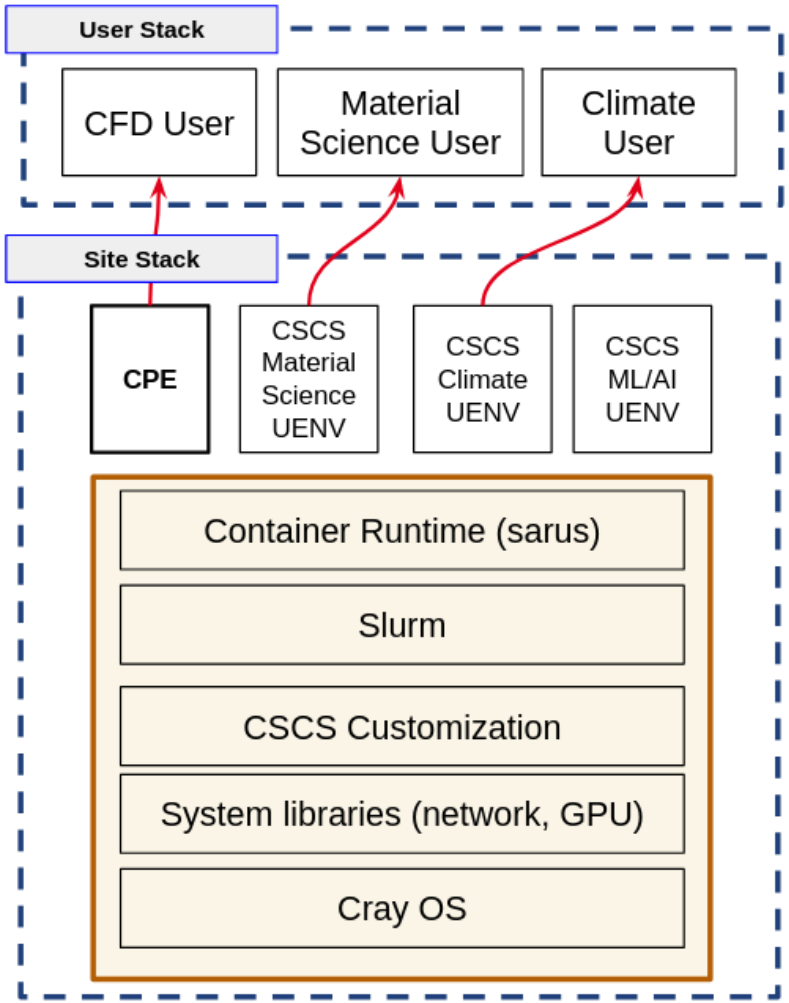
Start with a simpler foundation:

- CrayOS + libfabric + Slurm;
- no CPE;
- less frequent changes over a smaller area.

Provide workflow-specific software stacks:

- only the packages that are needed;
- deployed independently;
- built using Spack.

"You can create environments without modules using Spack. "
"It is tricky to configure Spack. "



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

The Stackinator: Building Environments

Stackinator is Opinionated

Self contained software stacks are built through a workflow codified in a tool CSCS developed Stackinator.

provide the inputs: YAML recipe and system configuration ...

perform the steps: stack-config then make ...

...to build a software stack following the best-practices and HPE Cray EX-specific methods developed by CSCS (Harmen Stoppels).

Each stack exposes a spack upstream interface, and optional modules and environment views.

`https://github.com/eth-cscs/stackinator`

Stackinator

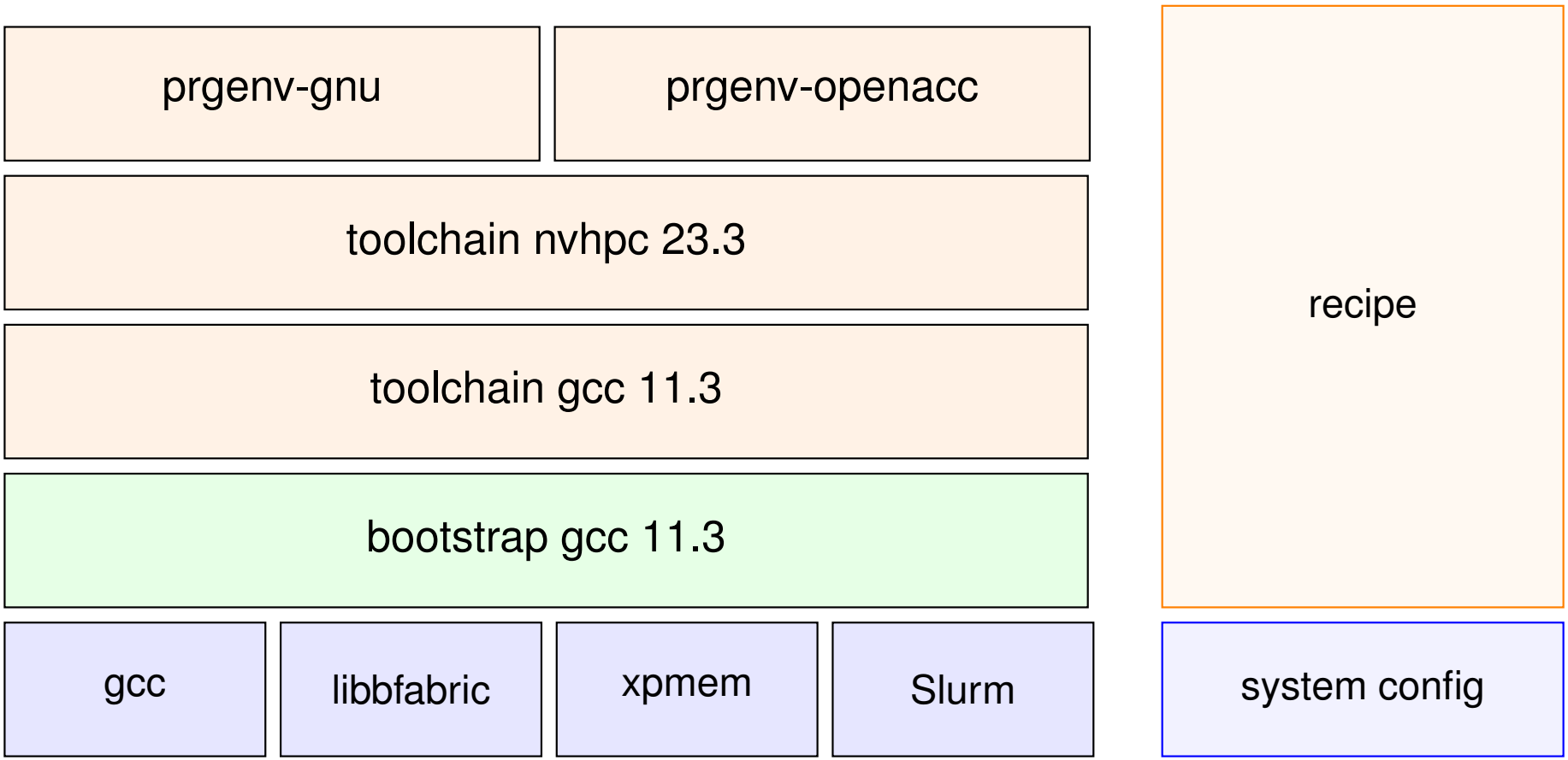
Stackinator provides a CLI tool to configure the software stack on the target system:

```
> stack-config --recipe $recipe_path \
    --system $CLUSTERNAME --build /dev/shm/build
> cd /dev/shm/build
> env --ignore-environment PATH=/usr/bin:/bin:`pwd`/spack/
    bin make store.squashfs -j64
```

- recipe: YAML files that describe compilers, software packages, and tests for software stack.
- system: System config for few libraries (gcc, libfabric, xpmem, slurm, rdma-core).
- mount: The installation path (in the recipe).
- build: Where the build will be performed.

Stacks

A “Spack Stack” is built in layers on top of a handful of external system dependencies.



System Configurations

A Spack configuration for the target vCluster that describes the handful of system dependencies.

compilers.yaml

```yaml
compilers:
- compiler:
  spec: gcc@7.5.0
   paths:
    cc: /usr/bin/gcc
    cxx: /usr/bin/g++
    f77: /usr/bin/gfortran
    fc: /usr/bin/gfortran
   flags: {}
   operating_system: sles15
   target: x86_64
```

packages.yaml

```yaml
packages:
 libfabric:
  buildable: false
  externals:
  - spec: libfabric@1.15.2.0
    prefix: /opt/cray/libfabric
    /1.15.2.0/
 slurm:
  buildable: false
  externals:
  - spec: slurm@22-5-2
    prefix: /usr
 xpmem: ...
 rdma-core: ...
```

Recipe: general configuration

Name, mount point, the version of Spack to use and mirror configuration.

config.yaml

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

Recipe: compiler toolchains

Compilers are built in three stages

1. bootstrap: gcc built using the system compiler (gcc 7.5.0).
2. gcc: Optimised gcc version(s) provided by the stack.
3. llvm: (optional) nvhpc and/or llvm toolchains buil with gcc from step 2.

compilers.yaml

```
bootstrap:
  spec: gcc@11
# gcc@11 languages=c,c++ build_type=Release ~bootstrap +
    strip
gcc:
  specs:
  - gcc@11.3
# gcc@11 build_type=Release +strip
llvm:
  requires: gcc@11.3
  specs:
  - nvhpc@22.7
# nvhpc@22.7~mpi~blas~lapack
```

Recipe: environments

environments.yaml

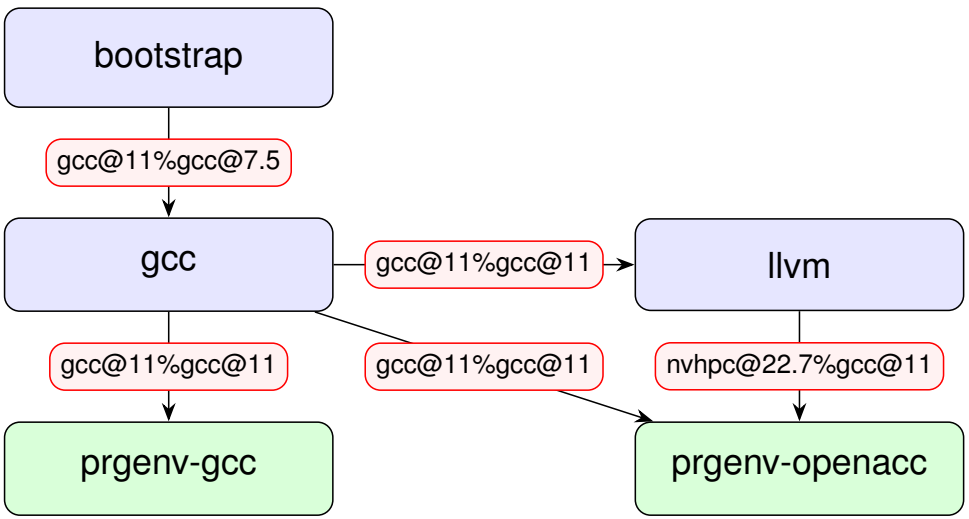
```yaml
prgenv-gcc:
  compiler:
    - toolchain: gcc
      spec: gcc@11
  unify: true
  mpi:
    spec: cray-mpich@8.1.18.4
    gpu: cuda
  specs:
  - cuda@11.8
  - osu-micro-benchmarks@5.9
  - openblas@0.3.21
  variants:
  - cuda_arch=80
  - +mpi
  - +cuda
```

```yaml
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
```

Building

stack-config generates a build path with a hierarchy of Makefiles that build the stack as a DAG of Spack environments.

- build path is in /dev/shm – build in memory
- Bubblewrap (bwrap) is used to mount the store path at the destination during builds




```
/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
```

Wait! What about MPI?

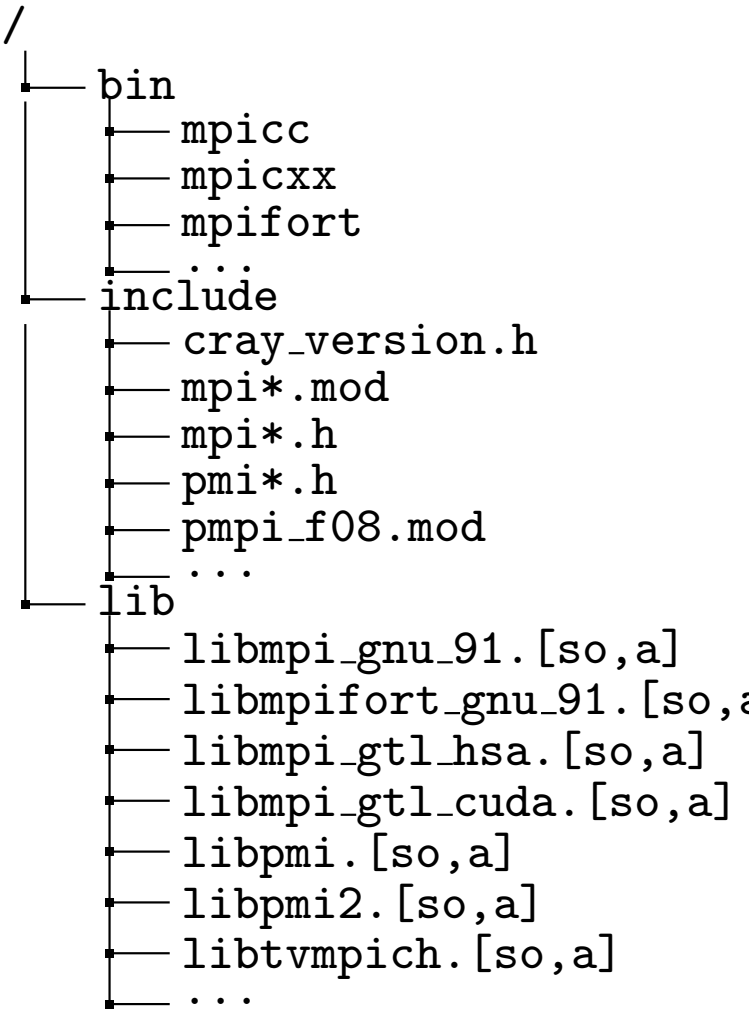
Cray-mpich is the only robust MPI for SS11 as of May 2023.

- We used OpenMPI+ucx on SS10

Stackinator uses a custom Spack package for cray-mpich:

- Repackage headers, libs, compiler wrappers from RPMs.
- Store as tar-balls CSCS-private artifactory.
- Run patchelf on libraries and string-substitution on compiler wrappers.

It takes an engineer an hour to create the binary package for each new CPE release.



MPI Configuration is Opinionated

environments.yaml: ”The user requests cray-mpich”

```
myenv:
    compiler:
    - toolchain: gcc
      spec: gcc@11
    mpi:
      spec: cray-mpich@8.1.18.4
      gpu: cuda
```

Generated Spack specs in spack.yaml

```
# cray-mpich specs are "simple"
specs:
- cray-mpich@8.1.18.4 +cuda

# The tool can generate more complex specs, e.g. OpenMPI
    on SS10:
specs:
- openmpi@4.0:4 +cuda +cxx +pmi schedulers=slurm fabrics=
    ucx
- ucx +rdmacm +cma +verbs +xpmem +ib_hw_tm +mlx5_dv +dc +
    ud +rc +dm +optimizations +gdrcopy ~assertions ~debug
```

Spack package

repo/packages/cray-mpich/package.py

```
@run_after("install")
def fixup_binaries(self):
  for root, _, files in os.walk(self.prefix):
    for f in [os.path.join(root, name) for name in files]
      if not self.should_patch(f): continue
      patchelf("--force-rpath", "--set-rpath", rpath, f)
      if "libmpi_gtl_cuda.so" in str(f):
        patchelf("--add-needed", "libstdc++.so", f)

@run_after("install")
def fixup_compiler_paths(self):
  filter("@@CC@@", self.compiler.cc,self.prefix.bin.mpicc)
  filter("@@PREFIX@@", self.prefix, self.prefix.bin.mpicc)
  if   "+cuda" in self.spec:gtl_library = "-lmpi_gtl_cuda"
  elif "+rocm" in self.spec:gtl_library = "-lmpi_gtl_hsa"
  else:                     gtl_library = ""
  filter("@@GTL@@", gtl_library, self.prefix.bin.mpicc)
```

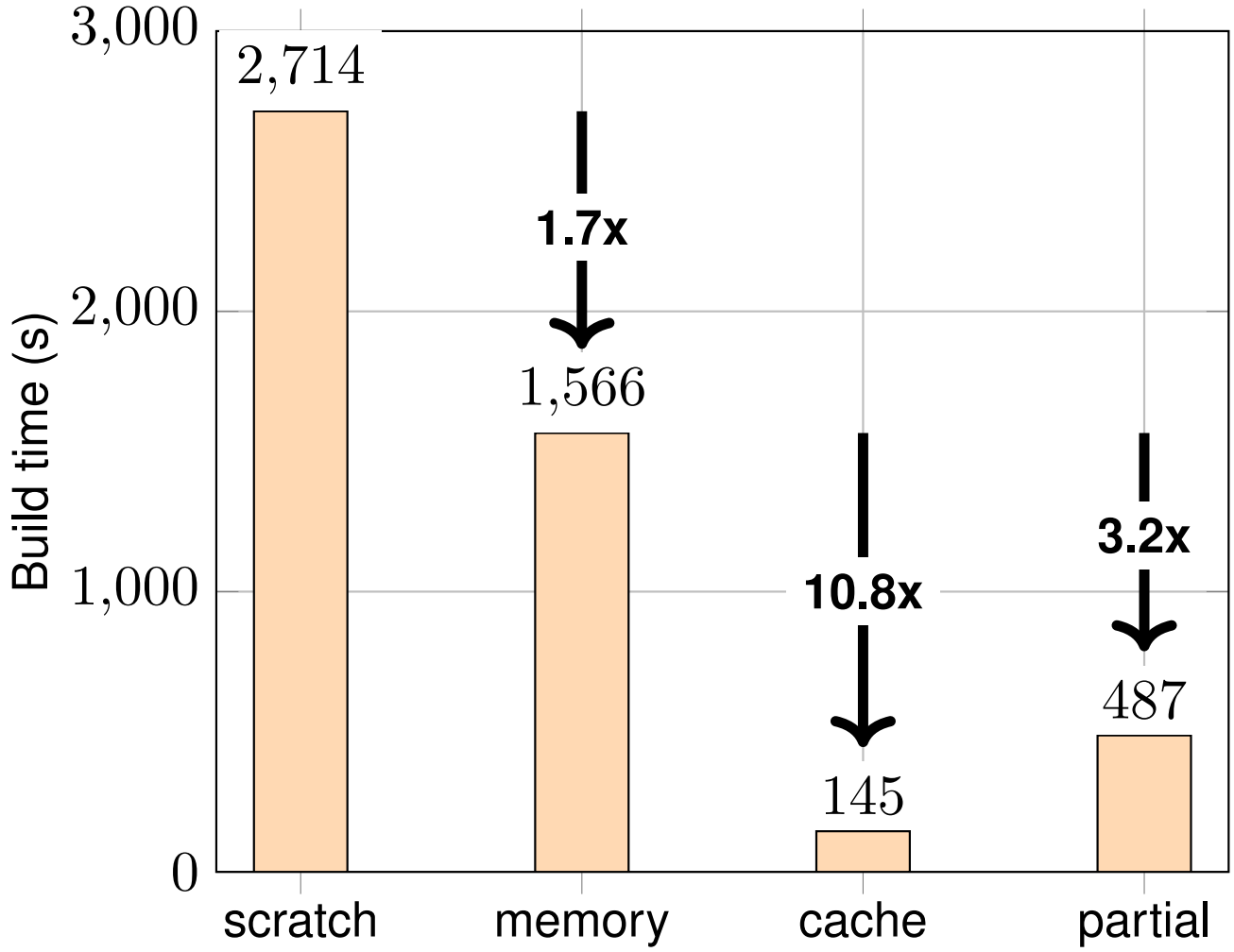
Optimising Build Times

Building stacks is resource intensive: 30 min – 3 hours with 64-cores.

Build times are the main pain point for developers.

- Parallelise the build: build Spack environments in parallel
  - Expose every opportunity to build packages concurrently.
- Build in memory:
  - Build in /dev/shm.
  - Use Bubblewrap (bwrap) to bind to the target installation path.
- Cache previous builds:
  - Only build packages once.
  - Use Spack binary build caches.

Optimising Build Times



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Deploying spack-stacks

SquashFS

The software stack can be copied to a shared file system once built.

At CSCS they are deployed as SquashFS images:

- consistent performance – always faster than a shared file system;
- reduced storage requirements – compression and deduplication.
- each stack is a single binary artifact – easy to version, roll back and manage in CI/CD pipelines.

SquashFS requires some additional tooling...

CLI Utilities

Non-privileged users are able to mount SquashFS images at runtime using the squashfs-mount CLI setuid utility that:

1. creates a new mount namespace;
2. mounts the SquashFS file through libmount;
3. then drops privileges and executes a given command.

mounting a squashfs image

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

The image is mounted in the new process – processes (users) on the same node can mount different images.

Open Source on GitHub with RPMs for Cray EX.
https://github.com/eth-cscs/squashfs-mount

SLURM

A Slurm plugin manages mounting environmnents on compute nodes.

Launch with explicit flags

```
% srun --uenv-mount=/user-environment \
       --uenv-file=img.squashfs \
       -n2 -N2 osu_bw
```

Inherit the environment from the login node

```
% squashfs-mount img.squashfs /user-environment bash
% srun -n2 -N2 osu_bw
```

Also works intuitively for sbatch – user can set a default image that, and individual srun in the script can use different environments.

Open Source on GitHub with RPMs for Cray EX.

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

CI/CD

CI/CD pipelines from recipe to deployed SquashFS image is a work in progress.

Recipes are stored in a GitHub repository – Pull requests and merges trigger a pipeline:

1. BUILD STAGE: launch a Slurm job on the target cluster+architecture that uses stackinator to configure then build the image.
2. push the generated image to a JFrog artifactory
3. TEST STAGE: pull the image and run a Slurm job that executes ReFrame tests.
4. post status to GitHub
5. DEPLOY STAGE: promote artifact to deployment artifactory (manual).

Results

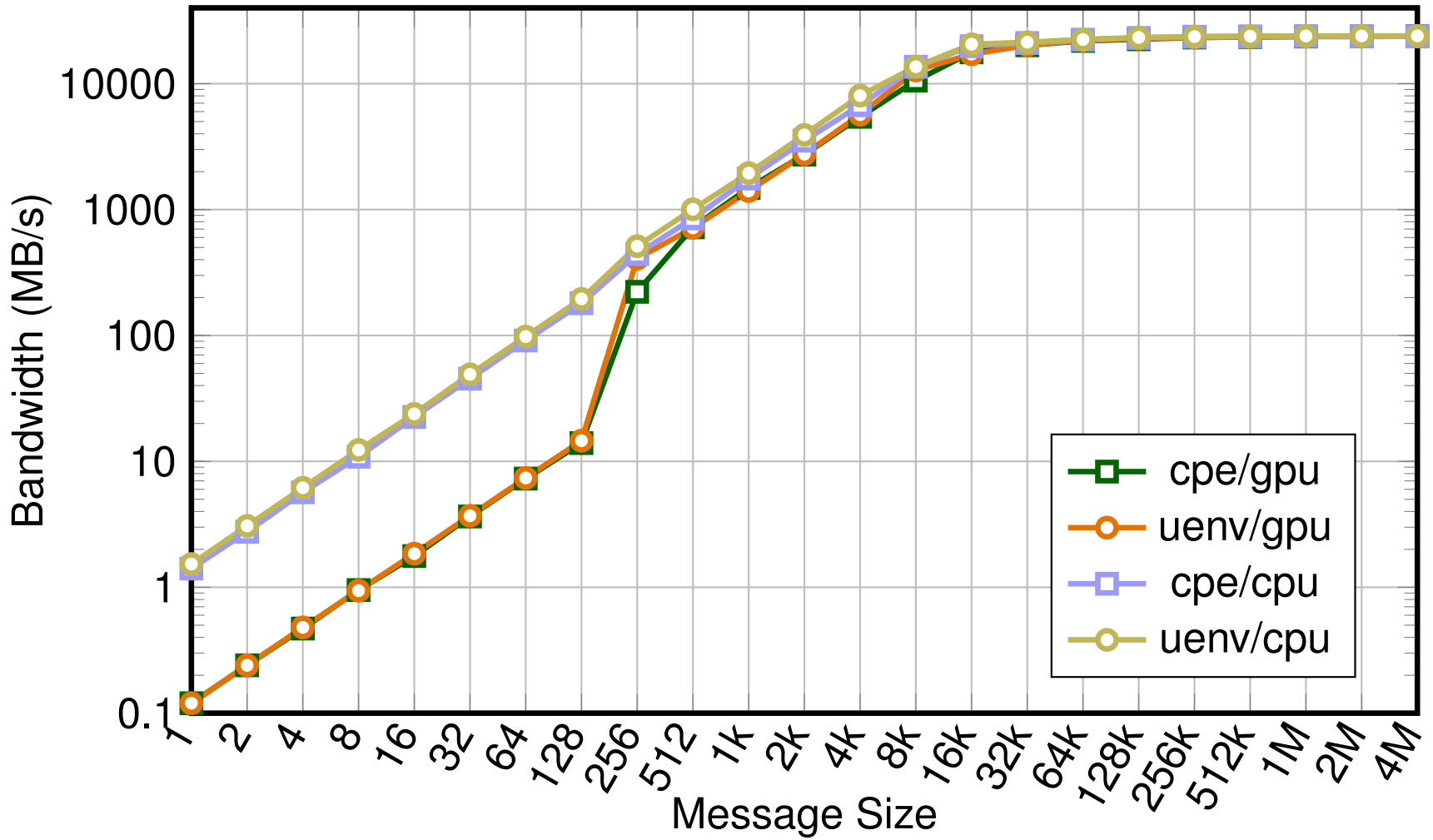
OSU

We run OSU benchmarks compiled using CPE and Spack Stacks to understand the effect of packaging cray-mpich outside CPE.

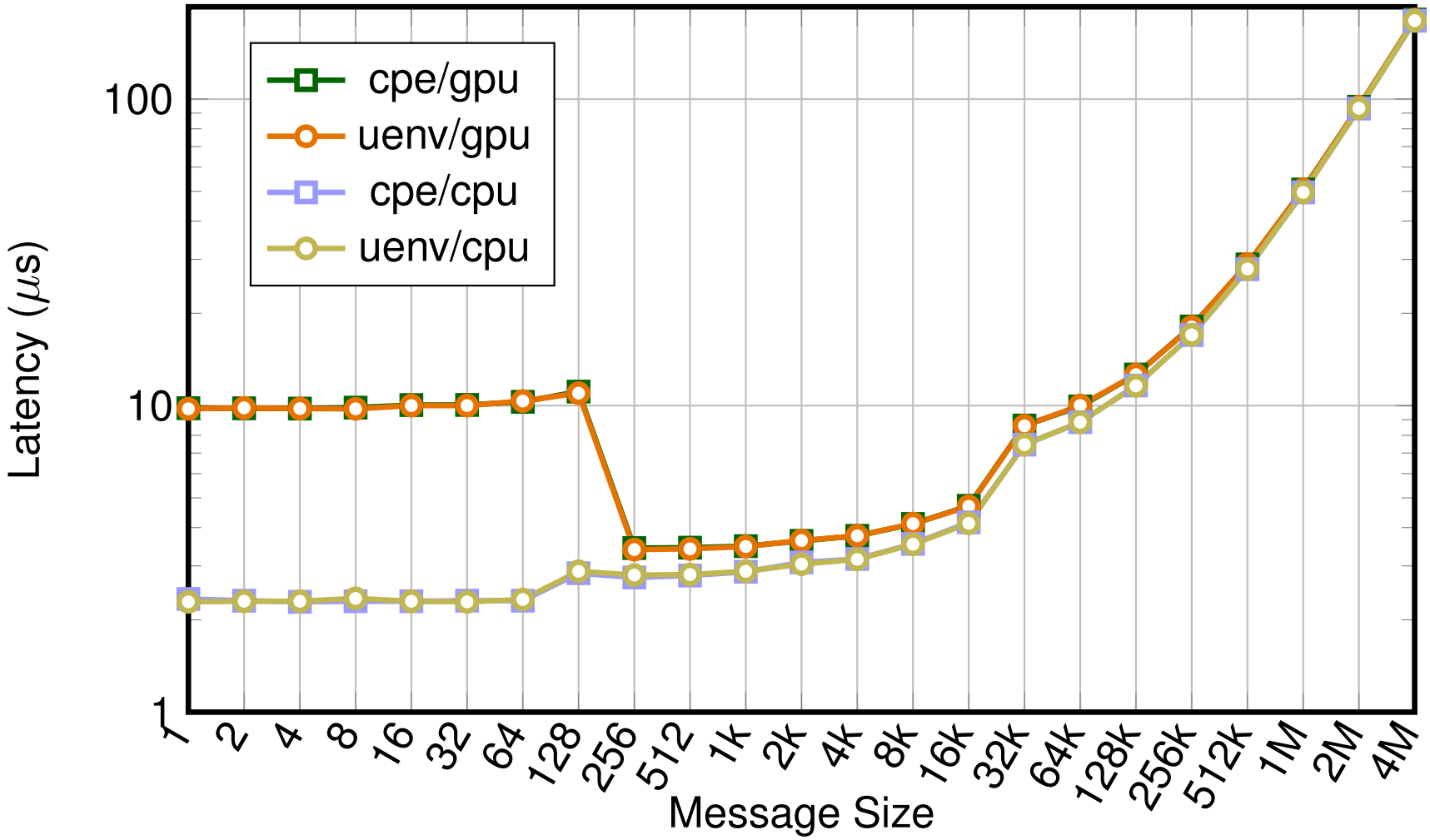
	CPE	Spack Stack
osu-benchmark	5.9	5.9
cray-mpich	8.1.21	8.1.24
gcc	11.2	11.3
cuda	11.6	11.8

The benchmarks are run on Clariden, a vCluster with 64-core EPYC CPU and 4 A100 GPUs – similar to Perlmutter.

OSU - P2P Bandwidth



OSU - P2P Latency



GROMACS

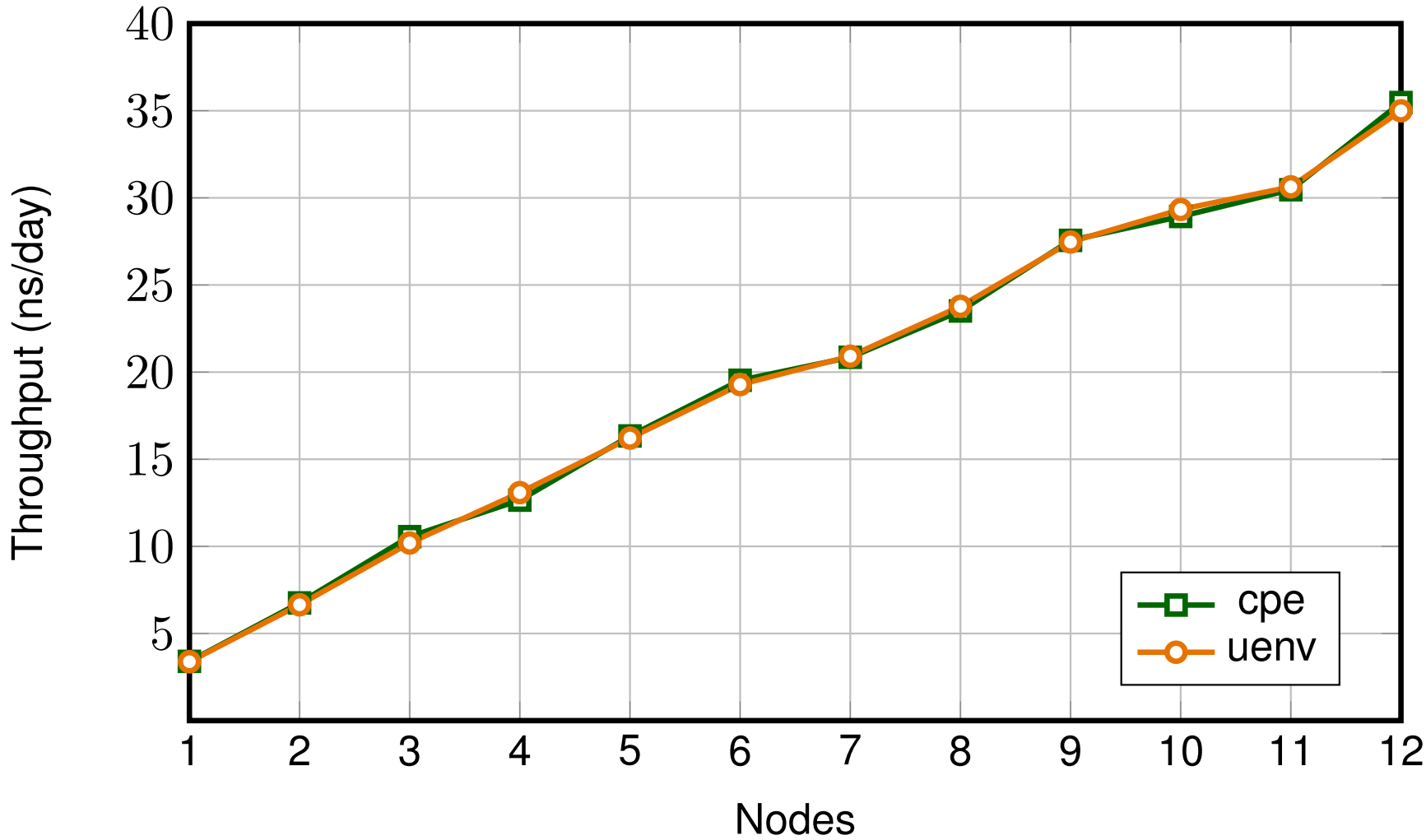
A GROMACS strong scaling benchmarks: a 1.4-million atom system (a pair of hEGFR Dimers of 1IVO and 1NQL) from the HECBioSim benchmarks suite.

	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

Run on Clariden, a vCluster with 64-core EPYC CPU and 4 Mi250x GPUs – identical to LUMI/Frontier/Setonix.

GROMACS - Strong Scaling

A difference of maximum ±1.5% between the CPE and the Spack-stack.



CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Wrapping up

An opinionated appeal

Integration of other CPE products – libsci, cce, etc – would be great.

Spack support is simple:

CPE packages can be installed individually without environment variables or modules, like normal software.

In an ideal world we could build cray-mpich and libfabric+CXI from source.

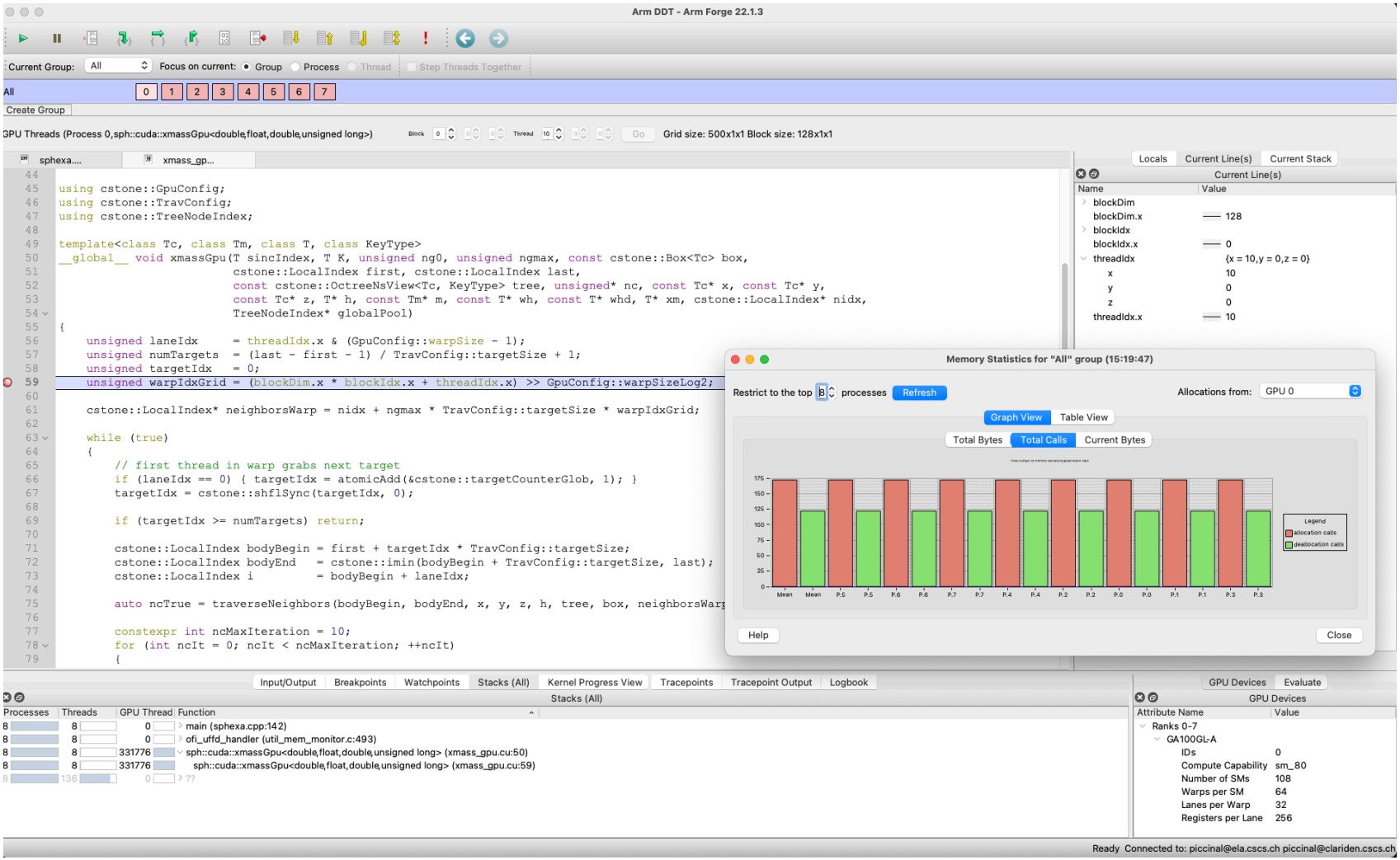
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Thank you

Backup

DDT


Arm DDT - Arm Forge 22.1.3
Current Group: All
Focus on current: Group Process Thread
Step Threads Together
Create Group
GPU Threads (Process 0,sph::cuda::xmassGpu<double,float,double,unsigned long>)
Grid size: 500x1x1 Block size: 128x1x1
sphexa....
xmass_gp...
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)
{
Locals
Current Line(s)
Current Stack
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)
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
Stacks (All)
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 ??
GPU Devices
Evaluate
GPU Devices
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

Configuration of mpicc

bin/mpicc

```
prefix="/user-environment/linux-sles15-zen3/gcc-11.3.0/
    cray-mpich-8.1.18.4-gcc-... long hash ..."
CC="${prefix}/bin/gcc"
$CC ${final_cppflags} ${final_cflags} ${final_ldflags} "${
    allargs[@]}" -I$includedir -L$libdir -Wl,-rpath,
    $libdir -lmpi -lmpi_gtl_cuda ${final_libs}
```

Objectives

We have the following objectives for our software stacks:

- Reproducable from simple recipes:
  - versionable with git;
  - descriptive: what not how.
- Separate system-specific configuration from recipe, so that the recipe does not need modification to
  - rebuild when a system is updated.
  - build for different systems.