Union of Containers: Layered Software Stacks with Cross-Container Dependency Reuse for HPC

Parmanand Sinha
University of Chicago Research Computing Center (RCC) · SC26 Research Poster

Validated on 2 clusters · 3 GPU architectures (sm_70, sm_80, sm_90) · 2 applications · 2 container runtimes

Motivation

- The stock E4S `cuda80` image is 32 GB and **1,826,534 filesystem entries**. Unpacking it on Anvil stopped at 69% with *Disk quota exceeded*: one image outgrew a 1.0M-file scratch quota.
- One 24-package GPU stack installs 10,772 files. A monolithic image repeats all of them for every `cuda_arch` target.
- Changing any single package forces a full rebuild and redistribution.
- GPU code built for one `cuda_arch` may not run on another: the image's `libkokkoscore.so` carries an sm_80 cubin and no PTX.

Idea: Union of Containers

Instead of one monolithic container, use two or more. A shared **provider** holds the architecture-generic dependencies and is built once. Each **specialized** container adds only what is specific to its variant and reuses the provider's packages rather than rebuilding them. At run time the dynamic linker resolves across both, so the application sees their union.

A variant can target its own architecture or framework without touching the shared base, and one provider can serve many specialized siblings.

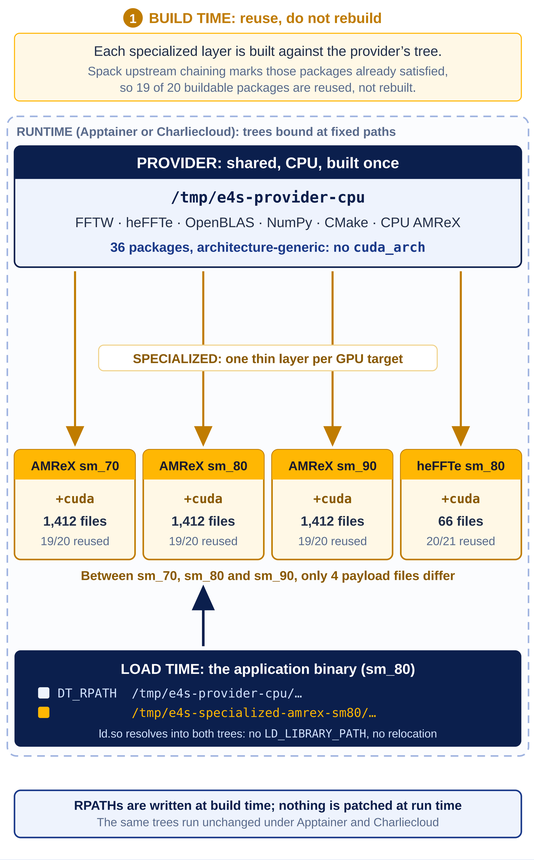
Layer	Role	This work's instance
Provider (shared)	Architecture-generic dependencies	36 packages: FFTW, heFFTe, OpenBLAS, NumPy, CMake, CPU AMReX, build tools
Specialized (per variant)	What only this variant needs	AMReX or heFFTe built `+cuda` for one `cuda_arch`
Base image	Externals, not rebuilt	gcc, glibc, MPICH, Python, CUDA

E4S supplies the recipes and Spack builds the layers, but the pattern is not tied to either: the Gentoo Prefix approach used by EasyBuild and the Digital Research Alliance of Canada builds the same kind of shared base, distributed via CVMFS.

Method & Testbed

- **Midway3** (UChicago): GPFS, Slurm, Tesla V100 (sm_70). Rocky 9.6 container on CentOS 8.4 hosts.
- **Anvil** (Purdue RCAC / NSF ACCESS): GPFS (IBM Spectrum Scale), A100 (sm_80) and H100 (sm_90). Ubuntu 24.04 container on Rocky 8.10 hosts.
- Pinned at both sites: Spack v1.2.1, `spack-packages` e4s-26.06.1, base image `e4s-cuda-26.06-cuda80`.
- **Monolithic comparator:** the identical 24-spec DAG, with the same AMReX hash, built as one tree.
- Files are `find -type f`; sizes are apparent bytes.

Architecture



Key Mechanism

Two requirements. (1) Binaries carry absolute `DT_RPATH` entries under both canonical roots, written at build time, so the dynamic linker resolves across containers with no `LD_LIBRARY_PATH` and no relocation. (2) The specialized build consumes the provider's installed packages as already-satisfied dependencies.

Any package manager that emits RPATH-linked install trees and can reuse packages across trees meets both. This work uses Spack's `shared_linking: rpath` and upstream chaining.

Placement rule: architecture-specific code lives only in the specialized layer; the provider stays arch-generic.

Resolution is done by `ld.so`, not the container runtime, so the same trees run unchanged under Apptainer and Charliecloud.

Result: Build Cost of One More GPU Target

1 of 20
buildable packages compiled (95.0% reused; heFFTe layer 20 of 21)

1.8–2.8×
faster build than rebuilding the monolith

Site, target	Monolithic	Layered	Speedup
Midway3, V100 sm_70	~13 min	~5 min	**2.6×**
Anvil, H100 sm_90	857 s	309 s	**2.8×**
Anvil, A100 sm_80	1,019 s	564 s*	**1.8×**

On sm_90 both arms ran the same day and AMReX alone took 4 m 20 s vs 4 m 19 s; the monolith also compiled FFTW for the benchmark (3 m 39 s), and without it the ratio is 2.1×. *The sm_80 layered figure is whole-job time, and its AMReX compile ran at half speed (8 m 44 s vs 4 m 23 s), so 1.8× understates the gap.

Result: Files on the Parallel Filesystem

Arrangement (Anvil, same 24-spec DAG)	Files
Monolithic, one target	10,772
Layered, one target	10,777
Layered, two targets (sm_70 + sm_80)	**12,189**
Monolithic, two targets	21,544

At one target, layering costs **5 files** (the second tree's `.spack-db` and `sbang`). Each further target adds 1,412 files instead of 10,772: **7.6× fewer**.

Across the sm_70, sm_80 and sm_90 layers, 1,397 of 1,412 files are byte-identical. **Only 4 files carry new content per architecture** (`libamrex_{1d,2d,3d}.a` and `AMReXConfig.cmake`) plus 11 Spack metadata files.

Findings for Practitioners

- **Use one Spack version for every layer.** A different Spack picks different default versions for some dependencies, so their hashes stop matching the provider's and everything is rebuilt. Pin the same Spack and `spack-packages` commit.
- **Put `upstreams:` inside `spack.yaml`.** Spack ignores a separate `upstreams.yaml` in the environment directory, and nothing warns you: the build just recompiles everything. Check with `spack config get upstreams`.
- **Give each container its own `/tmp`.** Two containers started at once that bind the same `/tmp` path collide creating the mount point (`mkdir: file exists`). `apptainer exec --contain` fixes it.
- **Build in `/tmp`, then copy to GPFS.** Spack's permission changes fail when installing straight onto a GPFS bind mount. Copy the finished tree to GPFS and bind it back at the same `/tmp` path at run time.

Result: Validation Matrix

Site	GPU	App	Runtime	Reused	GPU run
Midway3	V100 sm_70	AMReX	Apptainer	19/20	PASS
Anvil	A100 sm_80	AMReX	Apptainer	19/20	PASS
Anvil	A100 sm_80	AMReX	Charliecloud	same trees	PASS
Anvil	H100 sm_90	AMReX	Apptainer	19/20	PASS
Anvil	A100 sm_80	heFFTe	Apptainer	20/21	built

In every row, libraries installed in the trees resolved across them with `LD_LIBRARY_PATH` unset. The provider concretized to the same 36 packages, name for name, on both clusters. Charliecloud ran the Apptainer-built trees with no rebuild or relocation.

Result: Run-Time Cost

0.112 s
A100 benchmark: layered = monolithic = E4S-native

+26 ms
startup for binding a second tree

- **Compute:** AMReX MultiFab benchmark (256^3, 500 steps, 5 reps). Identical medians on A100, reproduced on two nodes; on H100, 0.0768 s layered vs 0.0761 s monolithic (within 0.9%).
- **Startup:** container start is about 0.29 s; the layered arm adds 26 ms (+8%) over a monolithic bind.
- **Sharing:** one provider tree served a GPU container and a CPU NumPy container at the same time (both alive in 7 of 14 samples), each with its own `/tmp`. Read-only sharing needs no locks.
- **Metadata:** first-touch listing of the 15,245-file provider on GPFS is 4.4× slower than warm (678 vs 153 ms), the same ratio seen on Midway3. The penalty is cold-start only.

Takeaways

Layering costs nothing at one target: 5 extra files and no measurable run-time difference. Every target after that is 1.8–2.8× cheaper to build and uses 7.6× fewer files. The result held on two clusters, three GPU architectures, two applications and two container runtimes.

Code, Spack environments & evidence logs
`github.com/pnsinha/union-of-containers`

This work was supported by the Better Scientific Software Fellowship Program, a collaborative effort of the U.S. Department of Energy (DOE), Office of Advanced Scientific Research via ANL under Contract DE-AC02-06CH11357 and the National Nuclear Security Administration Advanced Simulation and Computing Program via LLNL under Contract DE-AC52-07NA27344; and by the National Science Foundation (NSF) via SHI under Grant No. 2435328. This work used Anvil at Purdue University through allocation CIS261295 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by U.S. National Science Foundation grants #2138259, #2138286, #2138307, #2137603 and #2138296. It also used resources provided by the University of Chicago Research Computing Center.

Contact: Parmanand Sinha · University of Chicago Research Computing Center · pnsinha@uchicago.edu