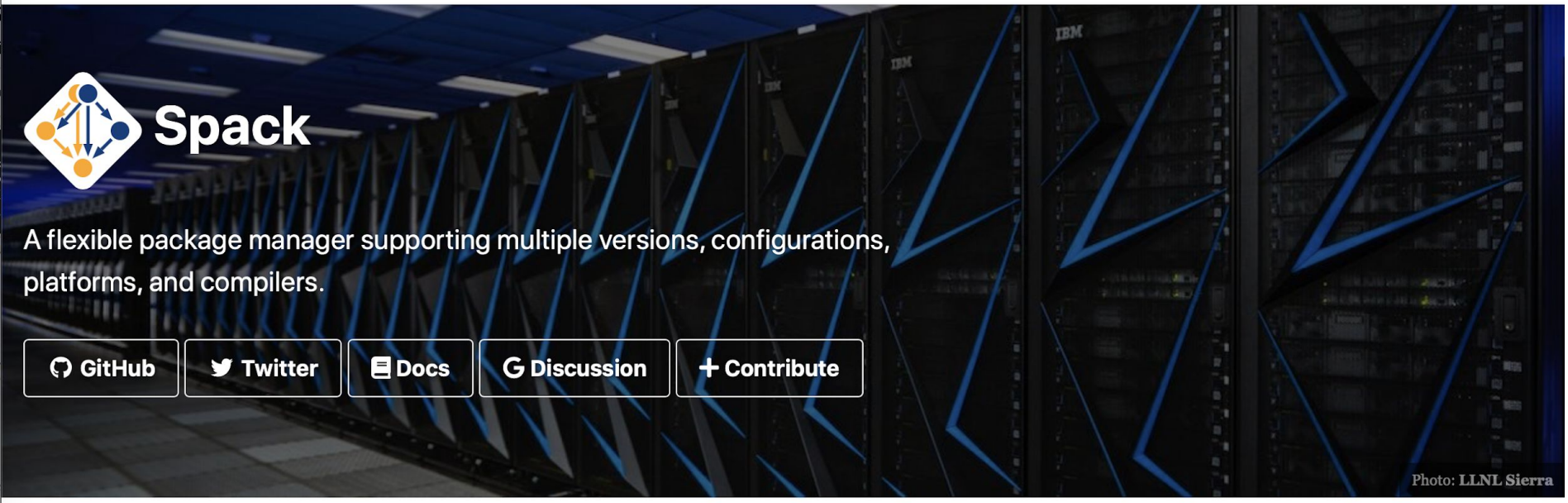
Spack for HEP Software Deployment

Ben Morgan

University of Warwick

With input from the HSF S&PT Working Group

2020-05-05



Welcome to Spack!

Spack is a package manager for supercomputers, Linux, and macOS. It makes installing scientific software easy. Spack isn't tied to a particular language; you can build a software stack in Python or R, link to libraries written in C, C++, or Fortran, and easily swap compilers or target specific microarchitectures. Learn more here.

https://spack.io

More on Spack

- **Far more than can be covered in this talk!**
- Spack has an excellent set of documentation available:
  - Full reference guide: https://spack.readthedocs.io/en/latest/
  - Tutorials: https://spack-tutorial.readthedocs.io/en/latest/#
- … with discussion fora on
  - Google: https://groups.google.com/forum/#!forum/spack
  - Slack: https://spackpm.slack.com
- … and development/issues/contribution on GitHub
  - https://github.com/spack/spack
- ***Here I'll focus only on a few specific aspects of Spack relevant for this meeting: microarchitecture handling, package deployment and reuse.***

Spack Specs

- Core concept in Spack is that every package has a "***spec***" that describes how it should be, or was, built, including
  - Version, compiler-version built with, options used ("***variants***")
  - Those parameters for each dependency used down the dependency graph
- Taking an example from the Spack docs:
  - `spack install mpileaks@1.2:1.4 %gcc@4.7.5 +debug -qt arch=bgq_os ^callpath@1.1 %gcc@4.7.2`
  - Means: "*install the mpileaks library at some version between 1.2 and 1.4 (inclusive), built using gcc at version 4.7.5 for the Blue Gene/Q architecture, with debug options enabled, and without Qt support. Additionally, it says to link it with the callpath library (which it depends on), and to build callpath with gcc at version 4.7.2*"
- ***Here we'll focus on how we can use specs handle microarchitectures/GPUs and how Spack organize sets of specs, optionally reusing existing system/Spack packages***

Microarchitectures

- Package specs have “***architecture***” attribute: `platform-os-target`
  - E.g. `linux-centos7-skylake`
- Aware of both generic families, e.g. `x86_64`, and specific implementations, e.g. `skylake` (Intel), `bulldozer` (AMD)
  - https://spack.readthedocs.io/en/latest/basic_usage.html#support-for-specific-microarchitectures
- Architecture is used in `install_path_scheme`, the directory layout template for installed packages, so they can be distinguished:

```
$ spack arch
linux-centos7-skylake
$ spack find
==> 2 installed packages
-- linux-centos7-ivybridge / gcc@8.3.0 --------------------------
zlib@1.2.11

-- linux-centos7-skylake / gcc@8.3.0 ----------------------------
zlib@1.2.11
$
```

NB: Can also have mix of platform/os in same spack instance. E.g. also distribute centos8, ubuntu, macOS through buildcaches

GPU Architectures (CUDA)

- Packages needing CUDA can write the `package.py` “recipe” using `CudaPackage`:
  - https://spack.readthedocs.io/en/latest/build_systems/cudapackage.html
- Introduces `cuda` and `cuda_arch` variants to enable use/select arch, e.g
  - `spack install vecgeom +cuda cuda_arch=72`

```
$ spack find -v vecgeom
==> 4 installed packages
-- linux-centos7-ivybridge / gcc@8.3.0 --------------------------
vecgeom@1.1.6 build_type=RelWithDebInfo ~cuda cuda_arch=none cxxstd=11 +gdml~gea
nt4~root+shared
vecgeom@1.1.6 build_type=RelWithDebInfo +cuda cuda_arch=72 cxxstd=11 +gdml~geant
4~root+shared

-- linux-centos7-skylake / gcc@8.3.0 ----------------------------
vecgeom@1.1.6 build_type=RelWithDebInfo ~cuda cuda_arch=none cxxstd=11 +gdml~gea
nt4~root+shared
vecgeom@1.1.6 build_type=RelWithDebInfo +cuda cuda_arch=72 cxxstd=11 +gdml~geant
4~root+shared
$
```

Environments: Grouping packages for use

- "*... used to group together a set of specs for the purpose of building, rebuilding and deploying in a coherent fashion*"
  - Manifest (`spack.yaml`) and Lock (`spack.lock`) files like Ruby/Rust (manual config also possible)
- Simplest use case: specs for specific versions+variants, e.g. for a release:

```yaml
# My example environment
spack:
  specs:
    - python@3.7.4 +shared
    - root@6.20.04
    - geant4@10.6.0 +qt +vecgeom
  view: true
```

- Environment will reuse packages that exist, install missing ones:
  - `spack env create myenv example.yaml`
  - `spack env activate myenv`
  - `spack install`
- Can be created in the Spack instance, or by end-user in separate directory

Using Spack Environments

- An environment can be activated/deactivated much like a `virtualenv`:

```
$ spack env activate -p example
[example] $ spack find
==> In environment example
==> Root specs
vecgeom

==> 4 installed packages
-- linux-centos7-skylake / gcc@8.3.0 ----------------------------
libiconv@1.16  veccore@0.6.0  vecgeom@1.1.6  xerces-c@3.2.2
[example] $ despacktivate
$
```

- It can contain a “view” of the packages just like an LCG view, or a “loads” shell script can be created that loads modulefiles for each package in the environment

Stacks: Environments for Librarians/Deployers

- Environments can also install a set of specs across different “axes” such as compilers or microarchitectures, e.g.

```
spack:
  specs:
    - matrix:
      - [cmake, zlib, vecgeom]
      - ['target=skylake', 'target=ivybridge', 'target=x86_64']
  view: False
```

  - *Results in 9 (+deps) installs, each of cmake, zlib, vecgeom compiled for each target*
  - *Much like build matrices in CI systems*
- Many more options available to conditionally include/exclude specs, and to “project” packages into views.

Containerize: Images from Environments

- Given an environment `spack.yaml` file, a recipe to pack it in a container image can be generated:
  - `spack containerize > Dockerfile`
- `container:` attribute in `spack.yaml` can be used to provide some fine tuning information, e.g.

```
spack:
  specs:
   - root@6.20.04
  container:
    format: docker   # Or singularity!
    os_packages:     # Additional system packages
      - HEP_OSlibs
```

- With HEP deployment use cases, more for those that need “fat” images or deployment of specific applications?

Buildcaches: Spack's binary distribution method

- A Spack instance may connect to a mirror containing **prebuilt packages**:
  - `spack mirror add mycache https://some.url`
- Packages are simple **GPG signed** tarballs organised under the mirror by **spec**
- Packages can be installed using
  - `spack install <spec>` : will look in buildcache first, only compiling if <spec> not found
  - `spack buildcache install <specs>` : install *matching* specs from buildcache, allows for install of multiple packages, potentially with different OS/microarchitectures than host machine.
- Binary packages can be added to a buildcache using
  - `spack buildcache create -d <buildcachedir> <spec>`
- Upcoming Spack developments will provide a LLNL hosted source/binary mirror plus a major update to the spec "concretizer" to speed up and better handle dependency resolution/reconciliation (see FOSDEM 2020 and SC19 presentations)

Packages.yaml: Reusing system packages

- This file allows customization of build preferences, and also to specify particular specs as provided externally to Spack, e.g.

```
1 packages:
2   openssl:
3     paths:
4       openssl@1.0.2k%gcc@4.8.5 arch=linux-centos7-x86_64: /usr
5     buildable: False
packages.yaml[+][yaml]                                          [5/5][1]
:
```

- Balance between system/Spack installed packages up to librarian
- ***Buildcaches*** can help here in reducing, albeit not eliminating, rebuilds for local installs (modulo package relocatability), reducing need for system packages

Chains: Reusing Spack installed packages

- A given spack instance can be pointed to the package install root of another using the `upstreams.yaml` configuration file, e.g.

```
upstreams:
  first:
    install_tree: /local/sspack-install/opt/spack
  second:
    install_tree: /cvmfs/some.repo.org/opt/spack
```

- Here, any required packages would be looked for in `first`, then second.
- ***However, upstreams have no knowledge that their packages are used***
  - A downstream must ensure that it does not use the upstream when the latter is being modified
  - An upstream should not remove packages as it does not know if a downstream depends on any of them
- Requires careful use and management.

Summary

- Spack provides a range of features that can assist in building and deploying software, especially across different architectures and systems
- Key feature is the ***architecture*** attribute of package ***spec***s
  - Automatically builds/organizes packages by OS, OS version, CPU family/implementation
- GPU (CUDA) arch also enters via ***cuda_arch variant*** for those packages that depend on CUDA
- Environments/Stacks allow sets of coherent packages to be defined, built, and deployed for use by end-users natively, or via a container image
- Buildcache capability provides signed binary packages
- Packages can be reused from OS or other Spack instance deployments
- ***Spack developers have been very open and helpful with HEP requirements, so further input and feedback on the librarian/deployment side welcome!***