# How to toolchain

Hands-on tutorial

## The problem

1. You want to compile & run your software on a bare metal machine
2. Make your toolchain due to requirements (version, components, etc.)
3. Probably want more toolchains (GCC, Clang, CUDA, SYCL)

# The goal of this tutorial

How to solve the problem, whithout losing your smile in the process

# Target tools

- `spack` to drive the compilation process
  - userspace, suggested way to compile stuff on CINECA
- `module` to manipulate the terminal environment
  - wildely used on supercomputers

# Compile your software

# Grab `spack` from its repository

- From a suitable working directory (e.g. `/opt`)

```
$ git clone https://github.com/spack/spack.git
$ source spack/share/spack/setup-env.sh
```

- All the stuff will be built & installed within the repo folder
  - configuration in `~/.spack`

# Bootstrap the environment

```
$ spack compiler find
==> Added 1 new compiler to /home/dgadioli/.spack/linux/compilers.yaml
    gcc@14.2.1
==> Compilers are defined in the following files:
    /home/dgadioli/.spack/linux/compilers.yaml
```

# Look for the C/C++ compiler

```
$ spack list gcc
```

- show all the available package(s) that have `gcc` in the name
- in this tutorial we will start from `gcc`

## Look for the target architecture

```
$ spack arch
```

- It will print the default target architecture for you local machine installation

# Inspect the configuration options for `gcc`

```
$ spack info gcc
```

- It lists all the available `version`s (14.2.0 -> 4.5.4)
- It lists all the available `variants` (e.g. w/ graphite, the languages, etc.)

# Look for packages

- Other than `info` you can also search online for available packages
- You can also navigate the packages offered by older version of spack

# See how spack concretize the installation

```
$ spack spec gcc
```

- It shows how spack will concretize the dependencies
  - `+` -> will be installed with spack
  - `e` -> imported from the environment (e.g. `glibc`)
- It reports also if spack is able to concretize the required version

# Build the compiler

```
$ spack install gcc@13 binutils=true graphite=true languages=c,c++,fortran
```

- the `@` it used to specify the software version
- the variants is a list of options with values

# Build the compiler (alternative syntax for `bool`)

```
$ spack install gcc@13+binutils+graphite languages=c,c++,fortran
```

- the `+` to enable a variant
- the `~` to disable a variant

# Install using a specific dependencies

```
$ spack install gcc@13+binutils+graphite languages=c,c++,fortran ^$DEP
```

- You can specify a specific dependency `$DEP` to install a package
- If you have already installed it you can use `^/$HASH`
- The same applies while using `spack spec` to see how it is concretize

# Install for other architectures

```
$ spack install gcc@13+binutils+graphite languages=c,c++,fortran target=$ARCH
```

- You can specify other target `$ARCH` for you installation

# List the installed package(s)

```
$ spack find gcc
```

list installed package by:

- Environment (OS + CPU family)
- Compiler used to compile the package
- `-l` -> long description with also the hash
- `-p` -> location path of the package

# Further Inspect the installed package(s)

```
$ spack spec /$HASH
```

- See how it has been concretized by spack
- You need the package `$HASH`

# *NOTE*: you need to find new compilers

```
$ spack load gcc@13
$ spack compiler find
```

# Use the new compiler to compile stuff

```
$ spack install gcc-runtime %gcc@13
$ spack install boost@1.87.0+fiber+graph+program_options %gcc@13
...
```

the `%` character specify which compiler it uses to compile it

# Uninstall packages

```
$ spack uninstall gcc@13
```

- It will uninstall the package
- If other packages depends on it spack will refuse to uninstall It
- You should use `--dependents` to remove also the dependent packages
- *ONLY AS A LAST RESORT* uninstall using `--force`

# Strong points

1. Automatically handle dependencies
2. Automatically handle building procedure
3. Unified way of configuring a package
4. Userspace, you can do it everywhere

# Weak points and caveats

- Strong software naming, different variants lead to different packages
  - use `$ spack find -l` to get the hash value
  - use `$ spack spec /clvldk7` to spec it
- Strong dependencies (related to hash, not package)
- Weird interface to list and load software
- Cannot handle custom `glibc` (for now)

# Manage your software

# Install environment module

```
$ spack install environment-modules
```

## Set up the `spack` and `module` integration

- generate module files for the installed package(s)
  - `$ spack module tcl refresh`
- enable automatic generation for future package(s)
  - `$ spack config add modules:default:enable:[tcl]`

# Initialize the module environment

- load the software package
    - `$ spack load environment-modules`
- enable the bash completition, `module` is a function :(
    - `. $(spack location -i environment-modules)/init/bash`

# Have fun with `module`

- list the available software: `module avail`
- load a software `module load mpich`
  - dependencies automatically handled
- unload a software `module unload mpich`
- unload everything `module purge`

# Manage software collections

- give the current set a name: `module save ligen_cuda`
- load a set: `module restore ligen_cuda`
- list all the sets: `module savelist`

# My `.bashrc` added lines

```
source /opt/dgadioli/spack/share/spack/setup-env.sh
spack load environment-modules
source $(spack location -i environment-modules)/init/bash
```

# Chaining spack

- Spack allows for multiple local spack installation
  - One shared spack with *system-level* software
  - *Per-user* spack installation for private software
- Modify you're `upstream.yaml` as defined here

```
upstreams:
  spack-instance-1:
    install_tree: /path/to/other/spack/opt/spack
    modules:
      tcl: /path/to/other/spack/share/spack/modules
```

## Modify install script

- Spack is nothing but a collection of python script
- If you don't like one you can modify It!
- You can find them under `spack/var/spack/repos/builtin/packages`
- Under each package there is a `package.py` which guide the package compilation

## Easy peasy lemon squeezy