CONSORTIUM FOR SMALL SCALE MODELING

Compiling the POMPA branch of COSMO for CPU and GPU

1. Introduction

The branch of COSMO developed in the POMPA project is a direct extension of COSMO. As compared to the official COSMO version, which requires only NetCDF and grib libraries for compilation, the POMPA branch of the code requires additional libraries in order to build. Furthermore, the build system used for these libraries is CMake instead of the usual Makefile. As a consequence some additional steps are required in order to successfully build COSMO. This document provides a background documentation in order to understand the build process and is complementary to a shell-script provided which details the steps required for the build.

2. Additional Libraries

COSMO requires a set of libraries in order to successfully link. Some libraries are required (i.e. non-optional) and some libraries can be activated via pre-processor options (e.g. -DMPI or -DRTTOV7). For the POMPA branch of COSMO, additional optional libraries are needed (see Figure 1).

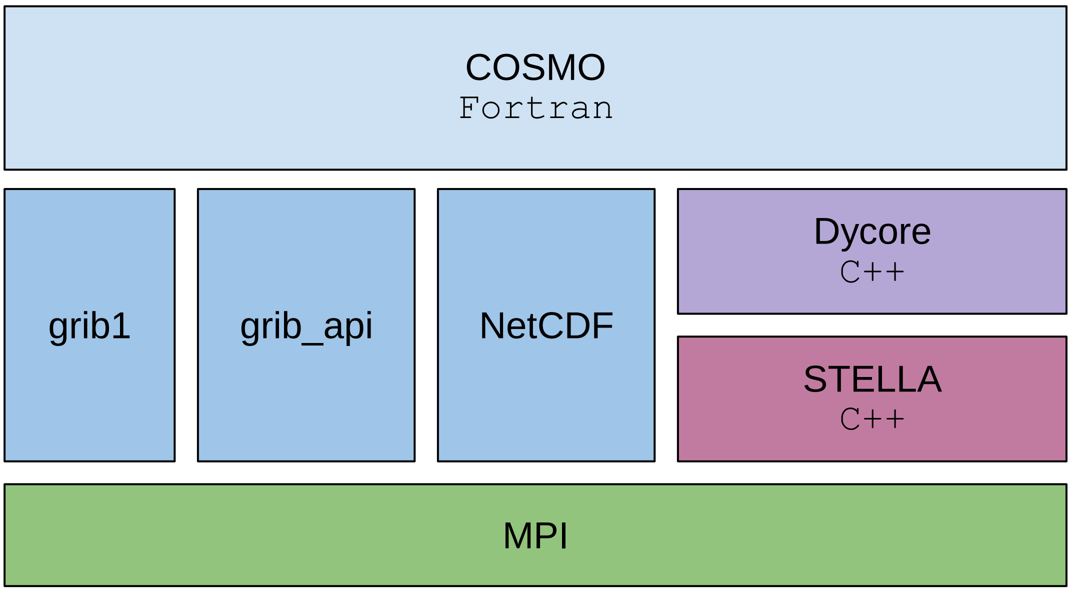

Figure 1 *Architecture of the POMPA branch of COSMO with the additional Dycore and STELLA components.*

C++ Dynamical Core

Within the POMPA project, the dynamical core has been rewritten in C++. The new dycore compiles into two libraries (**libDycore.a** and **libDycoreWrapper.a**), which contain the dynamical core and the interface of the dynamical core with the Fortran part of the code, respectively. In order to activate compilation with the C++ dynamical core, the pre-processor flag **-DCPP_DYCORE** has to be activated, otherwise the code will compile normally. The C++ dynamical core (or Dycore) exposes a C-level library to directly link with COSMO. In terms of linking this simply means that we simply need to tell COSMO the location of the C++ Dycore installation and give its corresponding linker flags:

```
-L<DYCORE_INSTALL_DIR>/lib -lDycoreWrapper -lDycore
```

Since the COSMO branch depends on the Dycore, the Dycore needs to be built and installed before building and installing COSMO. For building, the Dycore uses CMake, an industry standard that combines the features of autotools, instead of a set of complex Makefiles to make the build process as smooth as possible (`mkdir build; cmake ..; ccmake .`). CMake itself generates a Makefile based on user input and can be conveniently configured through either command line arguments or through an interactive text interface. The build can then be executed through `make install`.

STELLA

Internally, the Dycore uses the STELLA C++ library. STELLA is able to generate code for muls, Xeon Phi support is also planned. Since we are linking both STELLA and the Dycore dynamically we also need to expose STELLA to COSMO during the compilation of the POMPA branch of COSMO. This adds one lines to the linker:

```
-L<STELLA_INSTALL_DIR>/lib -lCommunicationFramework -ljson -lStella
-lgcl -lStellaUtils -lSharedInfrastructure -lstdc++
```

Internally, STELLA is built on top of a set of helper libraries (GCL, json, …) which are apparent on the linker line above. Since these are automatically built together with STELLA, the user does not have to explicitly worry about them.

Since the Dycore depends on STELLA, STELLA needs to be built and installed before building and installing the Dycore. Like the Dycore, STELLA is also built with CMake. As above, after running CMake, the build can then be executed through `make install`.

3. Configuration Options

From a COSMO point of view, STELLA and the Dycore are simply additional libraries like grib1 or RTTOV7. However, similar to RTTOV7, they have a set of options that can be enabled or disabled during compile time. It is important that these options are consistent between the COSMO build and the build of the libraries.

The POMPA branch of COSMO increases the number of configuration options for building and running COSMO.

- Debug or Release (aka Optimized) mode for compilation
- Single or double precision for floating point computations
- Compiling for CPU or GPU target
- Compiling with or without unit-tests which can be used to validate the build of the Dycore and STELLA

Often, it is important that these configuration options are set consistently across STELLA, the Dycore and COSMO. For example, for building a single precision COSMO executable, a version of the Dycore and STELLA has to be used which was also compiled for single precision.

The list of default CMake options of the Dycore for “standard” CPU and GPU builds are given below. Some of these default settings may change for special applications.

Dycore CMake Option	**CPU Build**	**GPU Build**
CMAKE_BUILD_TYPE	Release	Release
CUDA_BACKEND	OFF	ON
DYCORE_SHARED_POINTERS	OFF	OFF
DYCORE_TEST_SUBSET	OFF	OFF
DYCORE_UNITTEST	ON	ON
DYCORE_WRAPPER	ON	ON
ENABLE_CACHING	ON	ON
ENABLE_CUDA_STREAMS	ON	ON
ENABLE_OPENMP	OFF	OFF
ENABLE_PERFORMANCE_METERS	OFF	OFF
GCL	ON	ON
LOGGING	OFF	OFF
SHARED_LIBRARY	OFF	OFF
SINGLEPRECISION	<ON/OFF>	<ON/OFF>
STELLA_DIR	<STELLA_DIR>	<STELLA_DIR>

The list of default CMake options of STELLA for “standard” CPU and GPU builds are given below. Some of these default settings may change for special applications.

STELLA CMake Option	**CPU Build**	**GPU Build**
CMAKE_BUILD_TYPE	Release	Release
CUDA_BACKEND	OFF	ON

DISABLE_MERGE_STAGEs	OFF	OFF
ENABLE_CACHING	ON	ON
ENABLE_OPENMP	OFF	OFF
ENABLE_PERFORMANCE_METERS	OFF	OFF
GCL	ON	ON
GCLDEBUG	OFF	OFF
LOGGING	OFF	OFF
SINGLEPRECISION	<ON/OFF>	<ON/OFF>
STELLA_ENABLE_BENCHMARK	ON	ON
STELLA_ENABLE_COMMUNICATION	ON	ON
STELLA_ENABLE_SERIALIZATION	ON	ON
STELLA_ENABLE_TESTING	ON	ON
X86_BACKEND	ON	ON

When STELLA is built in GPU mode the X86 backend needs to be enabled. This then builds both a CPU and GPU libraries and their respective unit tests.

Dependencies

Default configuration

Both the Dycore and STELLA require and up-to-date C++ compiler and an installation of the Boost libraries. In contrast to Fortran, C++ libraries have a standard linker interface and it is thus not required to compile the libraries with the same compiler version and linker. However, for the best results we recommend an installation of GCC[1] 4.8+. Clang[2] is also known to work, however the code optimizer for GCC is known to producer faster code. For Boost[3] we tested versions between 1.49 and 1.55. In the current version we do not require any compiled variants of Boost. It is thus sufficient to supply the unpacked path of the Boost library to STELLA and the Dycore. In general, no configuration option is needed as CMake will automatically try to find the correct C++ compiler.

GPU configuration

To compile for GPUs a recent version (7.0+) of the NVIDIA Toolkit is required. CMake should be able to find the default installation paths of the NVIDIA Toolkit and it is simply required to enable the CUDA Backend in both STELLA and the Dycore. To avoid creating conflicts with GPU and CPU installations the compiled STELLA and Dycore libraries are

[1] https://gcc.gnu.org/
[2] http://clang.llvm.org/
[3] http://www.boost.org/

postfixed with a CUDA at the end of their filename. This changes the corresponding linker settings of the Dycore and STELLA in COSMO as following:

```
-LSTELLA_INSTALL_DIR/lib -lCommunicationFrameworkCUDA -ljson -
lStellaCUDA -lgcl -lStellaUtils -lSharedInfrastructureCUDA -lstdc++

-LDYCORE_INSTALL_DIR/lib -lDycoreWrapperCUDA -lDycoreCUDA
```

Additionally, the NVIDIA compiler needs to know the correct CUDA architecture for correct compilation, otherwise the code will not work on the machine. For example, the Kepler K80 GPUs provide the sm_37 architecture.

Note that if CPU and GPU linker lines are mixed the resulting error messages are hard to understand. The Fortran part of COSMO needs to be compiled with either PGI and Cray with the corresponding machine specific compiler flags. The POMPA project supplies default configurations for the CSCS Machines Piz Daint and Piz Kesch.

Single Precision configuration

By default, the Dycore and STELLA are built in double precision mode. To enable single precision the user needs to provide SINGLEPRECISION=ON to CMake of STELLA and the Dycore. We recommend installing the Dycore and STELLA to different directories when enabling single precision as single precision and double precision builds do not match and can generate confusing compiler errors. The COSMO Options file needs to be modified such that the -DSINGLEPRECISION flag is supplied. The single precision configuration can be configured on top of the GPU configuration.

4. Testing and Verification

Due to additional complexity of the code, each release is thoroughly tested in an automated manner. This stands apart from meteorological testing and focuses directly on the numerics. The code is distributed with various testing mechanisms.

- Testsuite: A small test program that runs a cosmo executable against a set of configurations and verifies the result against a double precision CPU reference. MeteoSwiss tests the code with the following combinations

COSMO Pompa version	CPU		GPU	
	Double	Float	Double	Float
Fortran version	threshold	threshold	-	-
Dycore Version	threshold	threshold	threshold	threshold

We enforce a match for double precision on CPU on the same architecture. For all other cases we verify the end result with a threshold.

- Dycore Unittests: Reference data from multiple test cases are generated from a special cosmo executable and the individual components are verified against a threshold of 1e-12 to make sure that we can match components of the dycore to their respective components in cosmo.
- STELLA Unittests: A test suite that verifies the inner workings of STELLA on all supported architectures. If a test of this suite is failing it is highly likely that the Dycore unittest and the cosmo testsuite is failing as well. Thus reducing the problem to a problem localized within STELLA.

Before a release is distributed the entire testsuite from STELLA, the Dycore and COSMO has to pass. This allows us to directly react to problems introduced by new code. This code is also distributed with the source code and allows and end user to verify correct operation on their machine as well.

5. Build Script

To make life easier we supply a script[4] which downloads, builds and installs the code with a recommended set of configuration options. We also try to avoid any user error. The script expects:

- COSMO_DEPDENCIES_DIR: A directory with the installed dependencies for COSMO, i.e. libgrib1, librigb_api, librttov7, libgrib1, libjasper, etc...
- INSTALLATION_DIR: A directory where everything should be installed
- CODE_DIR: A directory where the script is working in and the sources can be downloaded and compiled in.
- BOOST_ROOT: Path to the Boost root directory. Ideally this is already in the COSMO_DEPENDENCIES_DIR. Alternatively a boost version from the system can be supplied as well.
- ENABLE_SINGLE_PRECISION: Whether the single precision executable should be compiled or not. Default: OFF, to enable it, specify ON.
- ENABLE_CPP_DYCORE: To enable or disable the CPP Dycore. Default: ON, to disable it specify: OFF
- ENABLE_CUDA: Enables the GPU build. Default: ON, to disable it specify: OFF
- NVIDIA_CUDA_ARCH: Specifies the CUDA architecture. Talk to your computer vendor to get the correct setting.
- CC: The C compiler that should be used. Default: gcc
- CXX: The C++ compiler. Default: g++
- COSMO_COMPILER: Specifies the options file for cosmo. Default: cray. Alternative options: gnu, pgi
- BUILD_TYPE: Create a debug or a release executable. Default: Release for debug, use Debug
- ENABLE_COSMO_SERIALIZATION: Creates a special serialization build. This feature is recommended for advanced users. Default: OFF

[4] https://github.com/MeteoSwiss-APN/buildtools/tree/master/cosmo