LLVM_ENABLE_RUNTIMES=flang-rt

EuroLLVM 2025

Michael Kruse

Advanced Micro Devices GmbH

15th April, 2025

Outline

1 **Introduction**

2 **Legacy Flang Runtime**

3 **New Flang-RT**

4 **Remaining Work**

Outline

1 Introduction
- Usage
- The Mechanism
- Advanced Options

2 Legacy Flang Runtime

3 New Flang-RT

4 Remaining Work

LLVM_ENABLE_RUNTIMES?

Bootstrapping-Runtimes build

```
cmake ../llvm -GNinja                                 \
  "-DLLVM_ENABLE_PROJECTS=clang;lld;polly"            \
  "-DLLVM_ENABLE_RUNTIMES=compiler-rt;libc;libcxx;openmp"
```

LLVM_ENABLE_PROJECTS

- Compiled using host compiler (e.g. GCC)
- Same CMake build-dir as LLVM itself
- Intended to run on the host architecture

LLVM_ENABLE_RUNTIMES

- Compiled using just-built Clang
- Use separate CMake build-dir nested inside LLVM build-dir
- Intended to be used by binaries compiled by Clang
  - Can be a different architecture (cross-compilation)

How does it work?

CMake step

LLVM_ENABLE_PROJECTS

1. For each enabled project,
   `add_subdirectory(llvm-sourcedir/<project>)`

LLVM_ENABLE_RUNTIMES

1. Add build targets:
   `runtimes`, `install-runtimes`, `<runtime>`, `check-<runtime>`, `install-<runtime>`, …
2. For each target architecture,

```
llvm_ExternalProject_Add(runtimes ...)
```

which executes

```
cmake -GNinja                                             \
  -S llvm-sourcedir/runtimes                              \
  -B llvm-builddir/runtimes/runtimes-bins                 \
  -DLLVM_BINARY_DIR=llvm-builddir                         \
  -DCMAKE_{C,CXX}_COMPILER=llvm-builddir/bin/clang{++}    \
  -DCMAKE_{C,CXX}_COMPILER_WORKS=YES                      \
  "-DLLVM_ENABLE_RUNTIMES=<runtimes>"
```

1. `find_package(LLVM)`, `find_package(Clang)`
2. Find tools such as `llvm-lit`, `FileCheck` in `llvm-builddir`
3. For each enabled runtime,
   `add_subdirectory(llvm-sourcedir/<runtime>)`

How does it work?

Ninja step

LLVM_ENABLE_PROJECTS: `ninja <project>`

1. CMake ensured all necessary dependencies

LLVM_ENABLE_RUNTIMES: `ninja <runtime>`

1. Build dependencies such as `clang`, `FileCheck`, etc
2. Run configure step for runtimes
3. Build dependencies for runtimes
4. Execute

```
ninja -C llvm-builddir/runtimes/runtime-bins <runtime>
```

1. Build selected runtime

Runtime Options

Pass Options to Runtimes Build

```
cmake ...                                                   \
  -DCMAKE_CXX_FLAGS=-fmax-errors=1                          \
  "-DRUNTIMES_CMAKE_ARGS=-DCMAKE_CXX_FLAGS=-ferror-limit=1"
```

- `-fmax-errors=1` is for gcc
- `-ferror-limit=1` is for clang

Multiarch & Pass Arch-specific Options

```
cmake ...                                                          \
  "-DLLVM_RUNTIME_TARGETS=default;aarch64-linux-gnu"               \
  "-DRUNTIMES_aarch64-linux-gnu_CMAKE_CXX_FLAGS=-march=cortex-a57"
```

- Will create one builddir per target

Standalone-Runtimes build

CMake step

```
cmake -GNinja llvm-srcdir/runtimes     \
  -DLLVM_BINARY_DIR=llvm-builddir      \
  "-DLLVM_ENABLE_RUNTIMES=<runtimes>"
```

- Projects (LLVM, Clang, ...) compiled separately
- Uses default C/C++ compiler (e.g. gcc)

Ninja step

```
ninja <runtime>
ninja check-<runtime>
```

Outline

1 Introduction

2 Legacy Flang Runtime

- Usage
- The Problem

3 New Flang-RT

4 Remaining Work

Legacy Flang Runtime

flang/runtime/CMakeLists.txt

In-Tree build

- Like a LLVM_ENABLE_RUNTIMES build:

  ```
  add_subdirectory(runtime)
  ```

Standalone build

```
cmake llvm-srcdir/flang/runtime \
  -DLLVM_DIR=...                \
  -DCLANG_DIR=...               \
  -DMLIR_DIR=...

cmake llvm-srcdir/flang/lib/Decimal \
  -DLLVM_DIR=...                    \
  -DCLANG_DIR=...                   \
  -DMLIR_DIR=...
```

What is the Problem?

- Inconsistent with other LLVM runtimes
- For cross-compilation targets, must compile each target in standalone build
- GPU offloading: Build auxiliary target runtime separately
  - As done for openmp-offload, libc, compiler-rt, libcxx, ...
- Standalone build does not include `iso_fortran_env_impl.f90`
- Source code shared with Flang and Runtime
  - ABI assumed to be the same
  - Compile code and runtime code have different requirements

    E.g. runtime code must not link to C++ standard library
  - No clear separation which file belongs where
- Compiled binary shared with Flang and Runtime
  - Runtime built with different flags

    E.g. `-fno-lto`

Outline

1 Introduction

2 Legacy Flang Runtime

3 New Flang-RT
- Usage
- The Difficulties
- The Changes

4 Remaining Work

Building Flang-RT

Bootstrapping-Runtimes build

```
cmake  -GNinja ../llvm                      \
  "-DLLVM_ENABLE_PROJECTS=clang;mlir;flang" \
  "-DLLVM_ENABLE_RUNTIMES=flang-rt"
```

Standalone-Runtimes build

```
cmake -GNinja llvm-srcdir/runtimes                 \
  "-DLLVM_ENABLE_RUNTIMES=flang-rt"                \
  -DLLVM_BINARY_DIR=llvm-builddir                  \
  -DCMAKE_Fortran_COMPILER=llvm-builddir/bin/flang \
  -DCMAKE_Fortran_COMPILER_WORKS=YES
```

- Flang must built from the same git SHA1

  No ABI contract
- CMAKE_Fortran_COMPILER_WORKS because flang before the runtime is available cannot produce executables

Things that Must Continue Working

- Shared library
- Quad-precision `math.h` support
  - gcc `libquadmath`
  - Native `sizeof(long double) == 16` with `libm`
  - f128 suffix functions (like `sinf128`) in `libm`
- Conditional `REAL(16)` support in Flang
- Unittests
  - GTest and "non-gtest" testing framework
- Windows static `.lib`
  - LLVM emits libgcc-ABI function calls, requires `clang_rt.builtins.lib` at link-time
  - msvc ships `clang_rt.builtins-x86_64.a`, but not used by the driver (anymore)
- Experimental OpenMP-offload build
- Experimental CUDA build
  - With `clang -x cuda` and `nvcc`

Library Names

Old Library Names	New Library Names
`libFortranRuntime{.a,.so}`	`libflang_rt.runtime{.a,.so}`
`libFortranDecimal{.a,.so}`	
`libFortranFloat128Math.a`	`libflang_rt.quadmath.a`
`libCufRuntime_cuda_${version}{.a,.so}`	`libflang_rt.cuda_${version}{.a,.so}`

- Same scheme as Compiler-RT: `libclang_rt.<component>{.a,.so}`
- `libFortranDecimal` integrated into `libflang_rt.runtime`
    - Decided by RFC
    - `libFortranCommon` also used by Flang
    - Made `flang` depend on `libcudart.so`

The Big Move

Principles

- Split some headers into a compiler- and a runtime part
- Definitions to `flang-rt/lib/$component/*.cpp`
- Non-private headers to `flang-rt/include/flang-rt/$component/*.h`
- Files used by both, Flang (the compiler) and Flang-RT (the runtime), remain in `flang/`
- Move "`Common`" files only used by Flang (the compiler) to Support
  - Remaining shared components: `FortranDecimal`, `FortranCommon` (header-only), `FortranRuntime` (header-only), `FortranTesting`

Old	**New**
`flang/runtime/*.h`	`flang-rt/include/runtime/*.h`
`flang/runtime/*.cxx`	`flang-rt/lib/runtime/*.cpp`
`flang/runtime/Float128Math/*`	`flang-rt/lib/quadmath/*`
`flang/runtime/CUDA/*`	`flang-rt/lib/cuda/*`
`flang/include/flang/Runtime/*.h`	`flang/include/flang/Runtime/*.h`
`flang/include/flang/Common/*.h`[1]	`flang/include/flang/Support/*.h`
`flang/unittests/Evaluate/{fp-}testing.h`	`flang/include/flang/Testing/*.h`
`flang/lib/Common/*.cpp`	`flang/lib/Support/*.cpp`
`flang/unittests/Evaluate/{fp-}testing.cpp`	`flang/lib/Testing/*.cpp`
`flang/test/**/*`[2]	`flang-rt/test/**/*.cpp`

LLVM_ENABLE_PER_TARGET_RUNTIME_DIR

Library Location

- Old Flang Runtime:
  `${CMAKE_INSTALL_PREFIX}/lib/libflang_rt.runtime.a`
  - Clash in multiarch/cross-compile scenarios
- `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF`:
  `${CMAKE_INSTALL_PREFIX}/lib/clang/$version/lib/$os/libclang_rt.buildins-$arch.a`
  - Windows, Apple, AIX
- `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=ON`:
  `${CMAKE_INSTALL_PREFIX}/lib/clang/$version/lib/$triple/libclang_rt.builtins.a`
  - Became default for Linux in Clang 19 (Now also BSD, OS390)
  - Assumptions leaking into `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR=OFF` as well ☹

- Only the last supported for flang-rt
  - `${CMAKE_INSTALL_PREFIX}/lib/clang/$version/lib/$triple/libflang_rt.<component>{.a,.so}`
  - `LLVM_ENABLE_PER_TARGET_RUNTIME_DIR` ignored

Shared Library

- Old scheme: `BUILD_SHARED_LIBS=ON`
  - Requires a second standalone build
- New scheme: Build static+shared library at the same time using *object libraries*
  - Done by (almost) every other runtime

CMake Options

- `FLANG_RT_ENABLE_STATIC`
  - Default: `ON`
- `FLANG_RT_ENABLE_SHARED`
  - Default: `OFF`

    ld prefers `.so` over `.a`, enabling it would be a breaking change

Experimental GPU Target Support

CUDA

- **clang**: Compile everything with `-x cuda`
- **nvcc**: Treats everything as CUDA source
- Requires libcudac++ (libc++ for CUDA), cannot use `<variant>` or `<optional>`
- Declarations must be annotated with `__host__ __device__`

OpenMP

- Compile everything with `-fopenmp --offload-arch`
- Declarations must be annotated with *`#pragma declare target`*

- Annotations selected with preprocessor macro `RT_API_ATTRS`
- Results in a *fat library*
  - Host and device code in a single file
  - For AMD/OpenMP we would rather compile them separately
    - Multiarch library with device code looked up when launching kernels

Outline

1 Introduction

2 Legacy Flang Runtime

3 New Flang-RT

4 **Remaining Work**

- TODOs

To Do Items

- Flang is not (yet) a cross-compiler

  Sometimes assumes ABI of host platform, e.g. `sizeof(long)`
- Compile builtin modules in the runtimes build using CMake
  - OpenMP's modules as well
  - Per-target modules
- Flang's Quadmath support must not depend on LLVM build environment
- Multilib support
- Shared library location
- Library versioning

AMD