Flang Progress Update

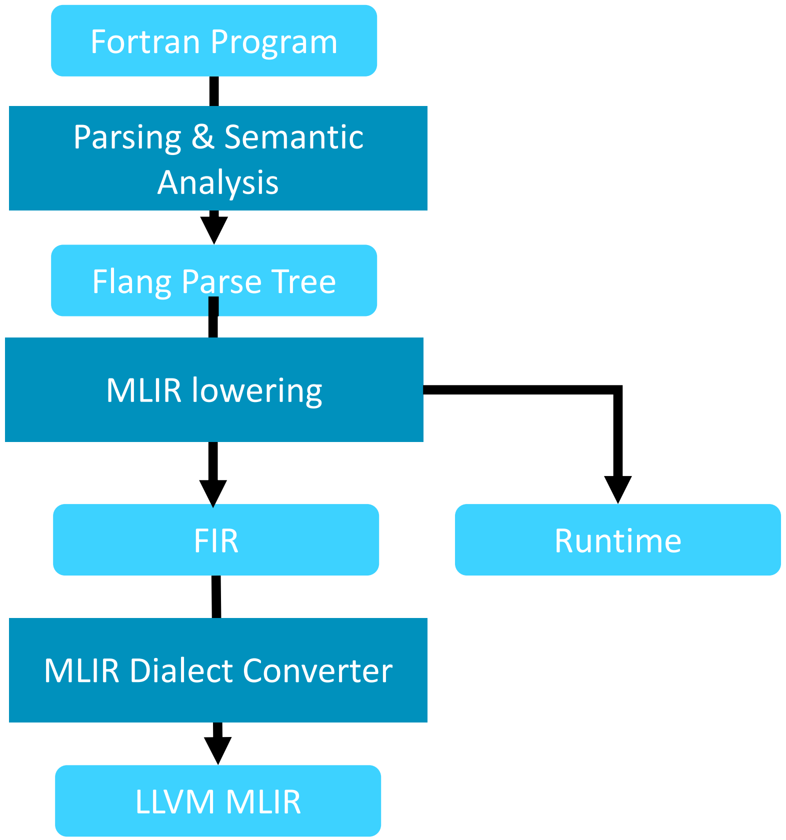
FOSDEM 2023

Kiran Chandramohan
4 Feb 2023

Contents

- Overview
- Story so far
- Status
- Development Efforts

Flang: Overview/High Level Flow



- New frontend developed from scratch
- Traditional Compiler Flow
  - LLVM based Fortran frontend
  - Generates LLVM IR
- Difference with Clang
  - Clang lowers from AST to LLVM IR
  - Has a high-level IR : Fortran IR (FIR)
- Uses MLIR infrastructure for FIR
  - MLIR interfaces with LLVM IR through the LLVM Dialect
  - FIR lowers to LLVM Dialect

Story so far

Euro LLVM 2018
News of a new LLVM based Fortran frontend at Nvidia supported by US DoE

April 2019
F18 project accepted as Fortran frontend of LLVM

April 2020
F18 merged in as llvm-project/flang

Parsing & Semantics (Fortran 2018), Runtime (Fortran 95) work in llvm-project

FIR and lowering work in f18-llvm-project Fortran 77, Fortran 95* (closing in), Upstreaming, Freeze (April 2022)

Merge into llvm (July 2022)

TODOs for known issues/unsupported features.Development of Fortran 2003+ features. Bug fixes. Performance work.

arm

Flang Current Status

- Not yet ready for general purpose use
  - Driver is temporarily called `flang-new`*
- Executables can be created
  - Use the `flang-experimental-exec` flag*
- Feature development for Fortran 95 standard is mostly complete
  - Issues error message for unimplemented features
- Development of Fortran 2003+ features in progress
- Testing
  - Using various commercial and free test-suites
  - Verified with SNAP, Cloverleaf, Spec rate
  - Continued testing with Spec speed, OpenRadioss etc

* Currently under discussion

Fortran Language support

Standard	Status
Fortran 77	Complete
Fortran 95	Complete (A few TODOs in character expression lowering, forall, and derived types with non-default lowerbounds)
Fortran 2003	Parser, Semantics and Runtime works modulo bugs In progress (Polymorphic types)
Fortran 2008	Parser, Semantics and Runtime works modulo bugs In progress (Block)
Fortran 2018	Parser, Semantics and Runtime works modulo bugs Not In Progress: Lowering, Codegen

Spec 2017 runtime – O3

Benchmark	Flang vs Gfortran-12	Flang vs Classic Flang	Comment
503.bwaves_r	0.93	1.02	
507.cactuBSSN_r	0.88	1.02	
521.wrf_r	1.06	1.11	
527.cam4_r	3.05	2.46	Array expression codegen
549.fotonik3d_r	1.74	1.76	Alias Analysis
554.roms_r	1.49	1.49	Array codegen, Alias Analysis
548.exchange2_r	2.82	1.30	Intrinsic Inlining, Function Specialization
	1.53	1.38	Geometric Mean

Flang is slower but catching up fast. A few months ago we were 2X vs gfortran.

Major development efforts

- Apologies for the efforts I am not highlighting
- HLFIR
- Polymorphic types (Fortran 2003)
- Performance
  - Alias Analysis
  - Assumed Shape Array Codegen
- OpenMP
  - Task, SIMD constructs
  - OpenMP offloading
- Driver

High Level FIR (HLFIR)

- FIR contains a minimal set of operations that models:
  - Fortran structured constructs, memory, allocation, descriptors
  - Big Gap between Fortran representation and FIR
- HLFIR carries more information into the IR
  - Enable optimisations
  - Provide better debug
- Introduces two concepts
  - Models expressions (that are not buffered)
  - Introduces Variables
- Initial lowering from parse-tree to HLFIR + FIR
  - Subsequently, HLFIR is converted to FIR
- For details see RFC by Jean Perier (Nvidia)
  - https://github.com/llvm/llvm-project/blob/main/flang/docs/HighLevelFIR.md

High Level FIR (HLFIR)

```
  integer :: y(2, 2) = reshape((/1, 2, 3, 4/), [2,2])
  integer :: s(2)
  s = SUM(y, DIM=1)
end
```

FIR

```
func.func @_QQmain() {
…
%1 = fir.alloca !fir.array<2xi32> {bindc_name = "s", uniq_name = "_QFEs"}
// lots of IR noise for putting together the arguments to the runtime call
…
// Runtime call allocates a buffer on the heap and returns the result in that
%16 = fir.call @_FortranASumDim(%12, %13, %c1_i32, %14, %c3_i32, %15) fastmath<contract> :
…
// copy from heap array to the real variable
…
// free heap allocated buffer
```


High Level FIR (HLFIR)

```
func.func @_QQmain() {
   %c2 = arith.constant 2 : index
   %0 = fir.alloca !fir.array<2xi32> {bindc_name = "s", uniq_name = "_QFEs"}
   %1 = fir.shape %c2 : (index) -> !fir.shape<1>
   // fortran variables (including attributes and debug info) are introduced to the IR with
   hlfir.declare
   %2:2 = hlfir.declare %0(%1) {uniq_name = "_QFEs"} : (!fir.ref<!fir.array<2xi32>>, !fir.shape<1>) ->
   (!fir.ref<!fir.array<2xi32>>, !fir.ref<!fir.array<2xi32>>)
   %3 = fir.address_of(@_QFEy) : !fir.ref<!fir.array<2x2xi32>>
   %c2_0 = arith.constant 2 : index
   %c2_1 = arith.constant 2 : index
   %4 = fir.shape %c2_0, %c2_1 : (index, index) -> !fir.shape<2> \
   %5:2 = hlfir.declare %3(%4) {uniq_name = "_QFEy"} : (!fir.ref<!fir.array<2x2xi32>>, !fir.shape<2>) ->
   (!fir.ref<!fir.array<2x2xi32>>, !fir.ref<!fir.array<2x2xi32>>)
   // minimal argument processing in hlfir
   %c1_i32 = arith.constant 1 : i32
   // sum operation returns a hlfir.expr, which is not yet buffered
   %6 = hlfir.sum %5#0 %c1_i32 {fastmath = #arith.fastmath<contract>} : (!fir.ref<!fir.array<2x2xi32>>,
   i32) -> !hlfir.expr<2xi32>
   // abstract array assignment saying that the unbuffered expression should be placed into this local
   variable
   hlfir.assign %6 to %2#0 : !hlfir.expr<2xi32>, !fir.ref<!fir.array<2xi32>>
   // if the expression is ultimately assigned a heap buffer, the fir.freemem would go here:
   hlfir.destroy %6 : !hlfir.expr<2xi32>
   return
}
```

Polymorphic Types

- Polymorphic types came in as part of Fortran 2003
- The types are only know at runtime
- Fortran has the class type for specifying such a type
  - It can refer to that type or any of its extended type
  - Extended type are types that inherits from other types in Fortran
- For details see RFC by Clement Valentine
  - https://github.com/llvm/llvm-project/blob/main/flang/docs/PolymorphicEntities.md

Polymorphic types

Fortran

```fortran
program mn
  type point
    real :: x, y
  end type point
  type, extends(point) :: point_3d
    real :: z
  end type
  type(point_3d) :: p3d
  call foo(p3d)
contains
subroutine foo(p)
  class(point) :: p
  select type (p)
   type is (point_3d)
      print *, "3d"
   type is (point)
      print *, "point"
  end select
end subroutine
end
```

FIR

```
func.func @_QQmain() {
  %0 = fir.alloca
!fir.type<_QFTpoint_3d{x:f32,y:f32,z:f32}> {bindc_name =
"p3d", uniq_name = "_QFEp3d"}
  %1 = fir.embox %0 :
(!fir.ref<!fir.type<_QFTpoint_3d{x:f32,y:f32,z:f32}>>) ->
!fir.class<!fir.type<_QFTpoint_3d{x:f32,y:f32,z:f32}>>
  %2 = fir.convert %1 :
(!fir.class<!fir.type<_QFTpoint_3d{x:f32,y:f32,z:f32}>>)
-> !fir.class<!fir.type<_QFTpoint{x:f32,y:f32}>>
  fir.call @_QFPfoo(%2) fastmath<contract> :
(!fir.class<!fir.type<_QFTpoint{x:f32,y:f32}>>) -> ()
  return
}
func.func @_QFPfoo(%arg0:
!fir.class<!fir.type<_QFTpoint{x:f32,y:f32}>>
{fir.bindc_name = "p"}) {
  fir.select_type %arg0 :
!fir.class<!fir.type<_QFTpoint{x:f32,y:f32}>>
[#fir.type_is<!fir.type<_QFTpoint_3d{x:f32,y:f32,z:f32}>>
, ^bb1,
 #fir.type_is<!fir.type<_QFTpoint{x:f32,y:f32}>>, ^bb2,
 unit, ^bb3]

…
```

Alias Analysis

- Alias information important for LLVM to do optimisations
- Aliasing rules in Fortran different from C
- Cannot directly reuse what is used for C
- In general, arrays do not overlap unless specified by pointer and target
  - Will benefit from the restrict patches
  - But the work is not yet complete
  - And there are issues with pointer escape
- For details see RFC by Slava
  - https://discourse.llvm.org/t/alias-analysis-in-llvm-flang/62639/42
- First step uses TBAA to distinguish descriptor access
  - For details see patch by Slava
  - https://reviews.llvm.org/D141820

Alias Analysis

```fortran
subroutine sb(a, b)
  integer :: a(:), b
  b = a(10)
end subroutine
```

LLVM MLIR with TBAA

```
module attributes {llvm.target_triple = "aarch64-unknown-linux-gnu"} {
  llvm.func @_QPsb(%arg0: !llvm.ptr<struct<(ptr<i32>, i64, i32, i8, i8, i8, i8, array<1 x array<3 x i64>>)>>
{fir.bindc_name = "a"}, %arg1: !llvm.ptr<i32> {fir.bindc_name = "b"}) {
    %0 = llvm.mlir.constant(0 : i64) : i64
    %1 = llvm.mlir.constant(9 : i64) : i64
    %2 = llvm.getelementptr %arg0[0, 0] : (!llvm.ptr<struct<(ptr<i32>, i64, i32, i8, i8, i8, i8, array<1 x
array<3 x i64>>)>>) -> !llvm.ptr<ptr<i32>>
    %3 = llvm.load %2 {llvm.tbaa = [@__flang_tbaa::@tag_4]} : !llvm.ptr<ptr<i32>>
    %4 = llvm.getelementptr %arg0[0, 7, 0, 2] : (!llvm.ptr<struct<(ptr<i32>, i64, i32, i8, i8, i8, i8,
array<1 x array<3 x i64>>)>>) -> !llvm.ptr<i64>
    %5 = llvm.load %4 {llvm.tbaa = [@__flang_tbaa::@tag_4]} : !llvm.ptr<i64>
    %6 = llvm.mul %5, %1  : i64
    %7 = llvm.add %6, %0  : i64
    %8 = llvm.bitcast %3 : !llvm.ptr<i32> to !llvm.ptr<i8>
    %9 = llvm.getelementptr %8[%7] : (!llvm.ptr<i8>, i64) -> !llvm.ptr<i8>
    %10 = llvm.bitcast %9 : !llvm.ptr<i8> to !llvm.ptr<i32>
    %11 = llvm.load %10 {llvm.tbaa = [@__flang_tbaa::@tag_5]} : !llvm.ptr<i32>
    llvm.store %11, %arg1 {llvm.tbaa = [@__flang_tbaa::@tag_5]} : !llvm.ptr<i32>
    llvm.return
  }
  llvm.metadata @__flang_tbaa {
    llvm.tbaa_root @root_0 {id = "Flang Type TBAA Root"}
    llvm.tbaa_type_desc @type_desc_1 {id = "any access", members = {<@root_0, 0>}}
    llvm.tbaa_type_desc @type_desc_2 {id = "any data access", members = {<@type_desc_1, 0>}}
    llvm.tbaa_type_desc @type_desc_3 {id = "descriptor member", members = {<@type_desc_1, 0>}}
    llvm.tbaa_tag @tag_4 {access_type = @type_desc_3, base_type = @type_desc_3, offset = 0 : i64}
    llvm.tbaa_tag @tag_5 {access_type = @type_desc_2, base_type = @type_desc_2, offset = 0 : i64}
  }
}
```

Codegen of Assumed Shape Array Arguments

- Assumed Shape Array arguments take the shape of the actual Arguments
- The actual Array passed can be strided
- For loops working on Assumed Shape Arrays
  - Code generated will have to consult the descriptor to use the stride to find successive elements
  - Use of the stride makes vectorisation difficult
  - Version the loop for stride == 1
- For details see patch by Mats Peterssen
  - https://reviews.llvm.org/D141306
  - https://discourse.llvm.org/t/transformations-to-aid-optimizer-for-subroutines-functions-with-assumed-shape-arguments/66447

Codegen Assumed Shape Array Arguments

Equivalent Code in Fortran shown

Input Code

```
do i = 1, n
  x(i * stride) = …
end do
```

After Versioning

```
if (stride == 1)
  do i = 1, n
    x(i) = ...
  end do
else
  do i = 1, n
    x(i * stride) = ...
  end do
end if
```

OpenMP

- Nearing OpenMP 1.1 completion
  - Due to a change in focus, this is delayed
  - Items to complete:
    - Privatisation
    - Atomic
    - Reduction
    - Detailed testing
- What is new?
  - Basic support for Task (OpenMP 3.0)
  - Clauses for SIMD construct
  - Some spec-2017 spec-speed benchmarks work now
    - cactuBBSN_s, wrf_s, roms_s, exchange2_s
- In progress
  - Target/Offloading, Task Dependencies, New loop related constructs (OpenMP 5.0+)

Driver

- Functional Driver based on the Clang Driver
  - Supports LLVM optimization pipelines
  - Can invoke Flang frontend plugins
- What is new?
  - Target specification, mcpu etc
  - fast-math and Ofast
  - MLIR/FIR optimisations added to the optimization pipelines
    - Arithmetic folding, TBAA generation, Intrinsic Inlining/Specialisation
  - LLVM pass plugins

+ In Progress
  - LTO, fsave-optimization-record, stack-arrays

Welcome to Contribute

- Contribute code
  - Follow the LLVM contributions process
- Report issues
  - https://github.com/llvm/llvm-project/issues
  - Labels: flang:build, flang:driver, flang:frontend, flang:ir, flang:runtime, flang
- Attend Flang Calls
  - Flang Community and Technical Calls biweekly
    - See google doc for details: https://docs.google.com/document/d/1Z2U5UAtJ-Dag5wlMaLaW1KRmNgENNAYynJqLW2j2AZQ/edit
  - Flang for OpenMP biweekly call
    - https://docs.google.com/document/d/1yA-MeJf6RYY-ZXpdol0t7YoDoqtwAyBhFLr5thu5pFI/edit

arm

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה