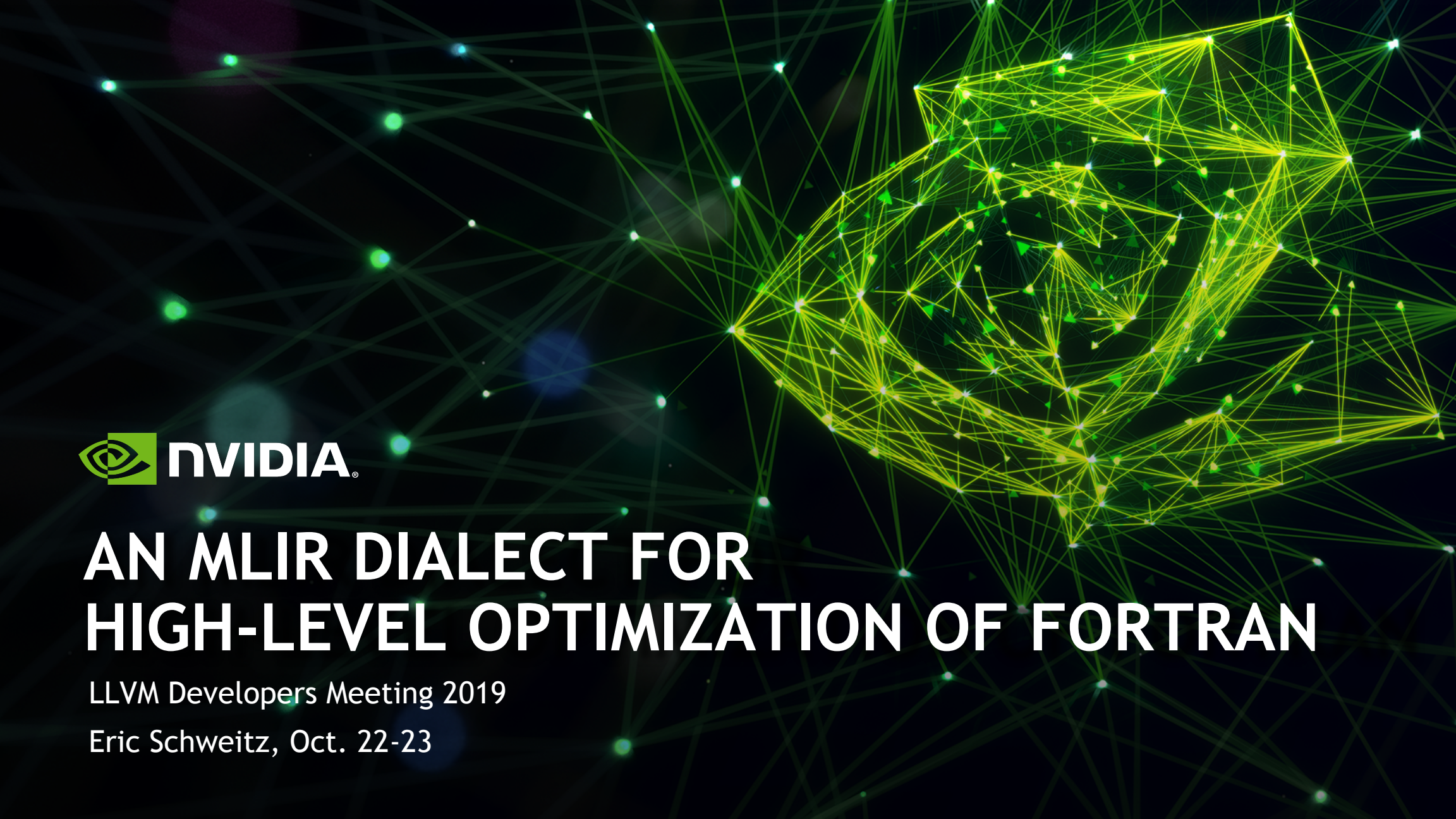


NVIDIA

AN MLIR DIALECT FOR HIGH-LEVEL OPTIMIZATION OF FORTRAN

LLVM Developers Meeting 2019

Eric Schweitz, Oct. 22-23

FLANG

The LLVM Fortran compiler

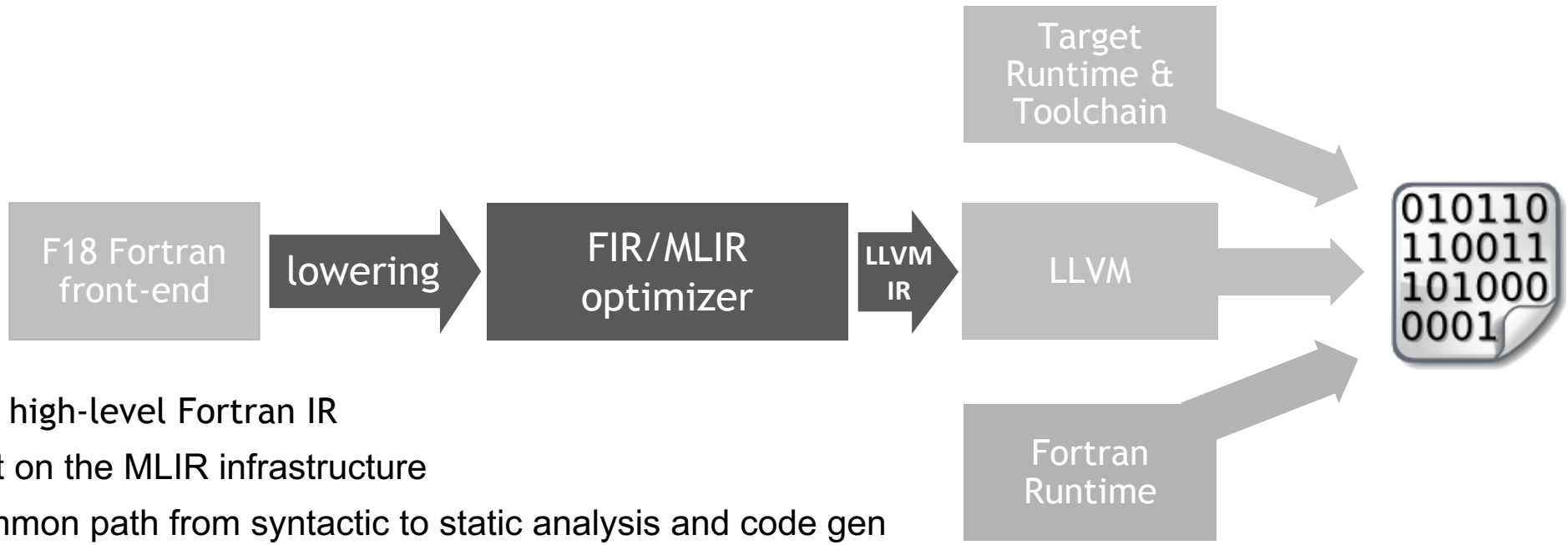
Fortran: the first high-level programming language, still important today

A dominant language in large scale simulation of physical systems
astrophysical modeling, hydrodynamics, molecular dynamics, materials science, electronic structure calculation, weather and climate models, etc.

The Flang open-source project is building a standards-compliant Fortran compiler

FLANG

The LLVM Fortran compiler



FIR: high-level Fortran IR

Built on the MLIR infrastructure

Common path from syntactic to static analysis and code gen

Shrink abstraction gap: core Fortran operational properties

Focus on writing Fortran aware optimizations

Separation of concerns: constraints checking vs. optimizing computation

MULTI-LEVEL IR (MLIR)

An infrastructure for building IRs

MLIR: an evolving unified infrastructure for developing IRs—from Tensorflow

MLIR approved for inclusion as an LLVM project on 10/7/2019

Extensible: compiler writers can add their own dialects, conversions, and optimization passes

Dialects: allow hybrid, co-existing representations and semantics

Structure similar to LLVM:

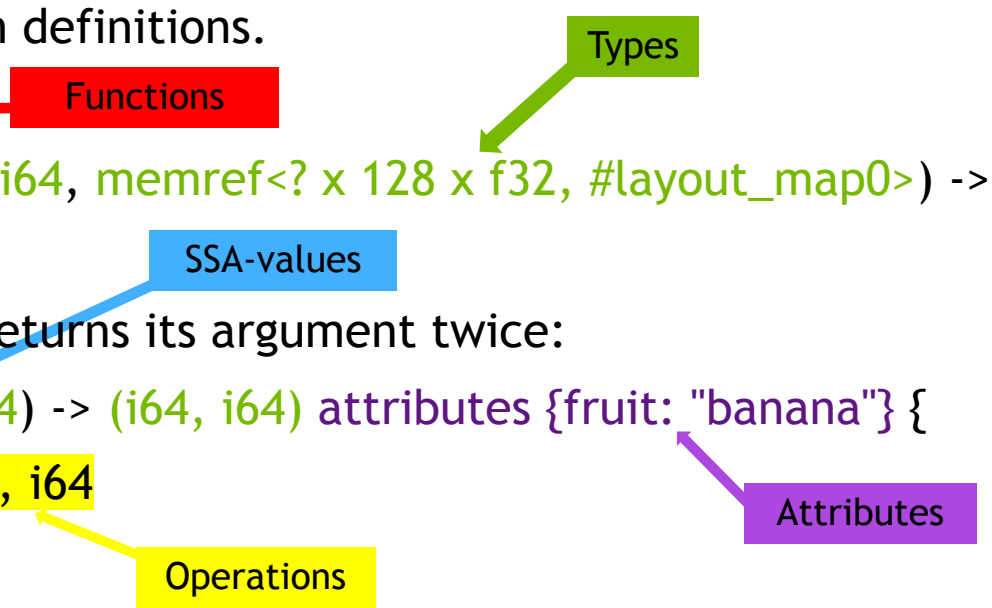
Modules, Functions, Basic Blocks, Operations, SSA-values, strongly typed, etc.

MLIR

An example (from MLIR Lang Ref)

```
// External function definitions.
func @abort()
func @scribble(i32, i64, memref<? x 128 x f32, #layout_map0>) -> f64

// A function that returns its argument twice:
func @count(%x : i64) -> (i64, i64) attributes {fruit: "banana"} {
  return %x, %x : i64, i64
}
```



FIR

The Fortran IR

Goal: Optimization of Fortran programs at a higher semantic level

A Fortran strength: array computation—multi-dimensional, dynamic index space, array ops

Dynamic objects: runtime parametric types and layout, default restricted aliasing

Object-oriented features: single inheritance, virtual dispatch, type-safe downcasting

Unique control-flow: internal subprograms, unordered loops, multiple entry and return-to

We want to be able to reason about
- different styles of loops and array structures
- generalized Fortran entity (object) types
- Fortran style OOP

FIR

Properties

Design Goals: an abstraction of Fortran

- smooth distinctions between type and attribute (FIR type system)
- memory-ssa to register-ssa support (alloca, undef, load, store)
- mechanics of deferred parameter properties (boxed types, embox, unbox)
- support runtime layout of types (record types, field and LEN parameter indices)
- dynamic dispatch of type-bound procedures (dispatch, dispatch tables)
- loop abstraction (loop and where)

FIR OPTIMIZATIONS

Some examples

- Loop transformations (vectorization)
- Array copy elimination
- Call specialization
- Devirtualization of calls

FIR OPTIMIZATIONS

Some examples

- Loop transformations (vectorization)
- Array copy elimination
- Call specialization
- Devirtualization of calls

LOOPS

An example of loop optimization

```
subroutine convolution(r, f, g)
  real :: r(:), f(:), g(:)        // assumed-shape vectors, from 1 to ?
  do n = 1, ubound(r, 1)
     do k = 1, ubound(g, 1)
        r(n) = r(n) + g(k) * f(n - k)
     end do
  end do
end subroutine convolution
```

LOOPS

An example of loop optimization

```
// subroutine convolution(r, f, g)
func @convolution(%r : !fir.box<!fir.array<?:f32>>, %f : !fir.box<...>, %g : !fir.box<...>) {
  %uf:3 = fir.box_dims %f, 0 : (!fir.box<...>, index) -> (index, index, index)  ... // and %ug:3
  fir.loop %n = 1 to %uf#1 {
    fir.loop %k = 1 to %ug#1 {
      %2 = subi %n, %k : index
      %3 = fir.coordinate_of %f, %2 : (!fir.box<...>, index) -> !fir.ref<f32>
      %4 = fir.load %3 : !fir.ref<f32> ... // and likewise %6 = load g[k]
      %7 = mulf %6, %4 : f32           ... // and likewise %9 = load r[n]
      %10 = addf %9, %7 : f32
      fir.store %10 to %8 : !fir.ref<f32>
}}}
```

LOOPS

Pseudo-code: post vectorization

```
// subroutine convolution(r, f, g)
func @convolution(%r : !fir.box<!fir.array<?:f32>>, %f : !fir.box<...>, %g : !fir.box<...>) {
  fir.loop %n = 1 to %uf#1 {
    %rn = fir.load %rp : !fir.ref<f32>
    fir.loop %k = 1 to %ug#1 step 4 {
      %v1 = “loadv” %fp : <4xf32>
      %v2 = “permutev” {to = <3,2,1,0>} (%v1) : <4xf32>
      %v3 = “loadv” %gp : <4xf32> ; %v4 = “mulvf” %v3, %v2 : <4xf32>
      %1 = “folding_addvf” %v4, %rn : (<4xf32>, f32) -> f32
      fir.store %1 to %rp : !fir.ref<f32>
}}}
```

FIR OPTIMIZATIONS

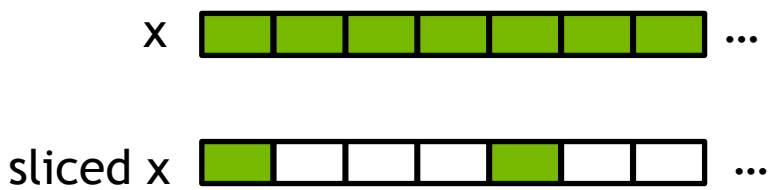
Some examples

- Loop transformations (vectorization)
- Array copy elimination
- Call specialization
- Devirtualization of calls

ARRAYS

Copy elimination

```
subroutine s1(x)
  real :: x(:)
  intent(in) x
  …
end subroutine s1
subroutine s2(x)
  real :: x(100)
  call s1(x(1:100:4))          ! pass an array slice
end subroutine s2
```



ARRAYS

Copy elimination

```
func @s1(%x : !fir.box<!fir.array<?:f32>>) {
  %dims:3 = fir.box_dims %x, 0 : (!fir.box<...>) -> (i32, i32, i32)
}
func @s2(%x : !fir.ref<!fir.array<?:f32>>) {
  %copy = fir.alloca !fir.array<25:f32> : !fir.ref<...>
  // copy %x.data to the buffer %copy          ! expensive, want to avoid
  %dims = fir.gendims(1,25,1) : !fir.dims<1>
  %xbox = fir.embox %copy, %dims : (...) -> !fir.box<...>
  fir.call @s1(%xbox) : (!fir.box<!fir.array<?:f32>>) -> ()
}
```

ARRAYS

Copy elimination

```
func @s1(%x : !fir.box<!fir.array<?:f32>>) {
  %dims:3 = fir.box_dims %x, 0 : (!fir.box<...>) -> (i32, i32, i32)
}
func @s2(%x : !fir.ref<!fir.array<?:f32>>) {
  %xref = fir.box_addr %x : (!fir.box<…>) -> !fir.ref<…>
  %dims = fir.gendims(1,100,4) : !fir.dims<1>
  %xbox = fir.embox %xref, %dims : (...) -> !fir.box<...>
  fir.call @s1(%xbox) : (!fir.box<!fir.array<?:f32>>) -> ()
}
```

FIR OPTIMIZATIONS

Some examples

- Loop transformations (vectorization)
- Array copy elimination
- Call specialization
- Devirtualization of calls

PROCEDURE CALLS

Specialization of CALL statements

```
subroutine s1(x)
  real :: x(:)        ! again, x is assumed-shape argument
  do i = 1, 10        ! but bounds are constants
    x(i) = f(i)
  end do
end subroutine s1

call s1(array)
```

PROCEDURE CALLS

Specialization of CALL statements

```
func @f(i32) -> f32
func @s1(%x : !fir.box<!fir.array<?:f32>>) {
  fir.loop %i = 1 to 10 {
    %1 = fir.coordinate_of %x, %i : (!fir.box<!fir.array<?:f32>>, i32) -> !fir.ref<f32>
    %2 = fir.call @f(%i) : (i32) -> f32
    fir.store %2 to %1 : !fir.ref<f32>
  }}

%box = fir.embox %array, %dims : (!fir.ref<!fir.array<?:f32>>, !fir.dims<1>) -> !fir.box<…>
fir.call @s1(%box) : (!fir.box<!fir.array<?:f32>>) -> ()
```

PROCEDURE CALLS

Specialization of CALL statements

```
func @f(i32) -> f32
func @s1.1(%x : !fir.ref<!fir.array<?:f32>>) {
  fir.loop %i = 1 to 10 {
    %1 = fir.coordinate_of %x, %i : (!fir.ref<!fir.array<?:f32>>, i32) -> !fir.ref<f32>
    %2 = fir.call @f(%i) : (i32) -> f32
    fir.store %2 to %1 : !fir.ref<f32>
  }}

%box = fir.embox %array, %dims : (!fir.ref<!fir.array<?:f32>>, !fir.dims<1>) -> !fir.box<...>
fir.call @s1.1(%array) : (!fir.ref<!fir.array<?:f32>>) -> ()
```

FIR OPTIMIZATIONS

Some examples

- Loop transformations (vectorization)
- Array copy elimination
- Call specialization
- Devirtualization of calls

OBJECT-ORIENTED PROGRAMMING

Fortran: Devirtualization

```
 type, extends(t) :: u
   contains
     procedure :: method => u_method
…
 type(u) :: uv
  call uv%method
```

OBJECT-ORIENTED PROGRAMMING

FIR: Devirtualization

```
// dispatch table for type(u)
fir.dispatch_table @dtable_type_u {
  fir.dt_entry "method", @u_method
}

  %uv = fir.alloca !fir.type<u> : !fir.ref<!fir.type<u>>
  fir.dispatch "method"(%uv) : (!fir.ref<!fir.type<u>>) -> ()
```

OBJECT-ORIENTED PROGRAMMING

FIR: Devirtualization

```
// dispatch table for type(u)
fir.dispatch_table @dtable_type_u {
  fir.dt_entry "method", @u_method
}

  %uv = fir.alloca !fir.type<u> : !fir.ref<!fir.type<u>>
  fir.call @u_method(%uv) : (!fir.ref<!fir.type<u>>) -> ()
```

CONCLUSION

Flang → FIR → LLVM

FIR is a dialect of MLIR, middle layer sitting between semantics checking and LLVM

Defines a minimalist, strongly-typed, high-level IR for Fortran sources

Suitable for Fortran-centric optimizations

Leverage MLIR work, since MLIR provides

- Easy-to-use, extensible framework, pretty parsers and printers, LLVM code gen
- Reuse optimizations: loop transformations, inlining, etc.

Part of Flang/F18 open-source project


NVIDIA