# Automatic Parallelization and OpenMP Offloading of Fortran

Ivan R. Ivanov[1,2], Jens Domke[2], Endo Toshio[1], Johannes Doerfert[3]

1. Tokyo Tech
2. RIKEN
3. LLNL

# Why Fortran?

Still widely used in the scientific community.

Tons of legacy code.

(Amount of fortran code running on HPC systems comparable to C and C++)

Has some nice features for scientific programmers. (e.g. native multi-dimensional arrays)

# Background

# Fortran array notation: easily operate on whole arrays

```fortran
real, dimension(n, n) :: x, y, z

y(1:n/2:2,1:n) = 0.1
y = sqrt(y * x) * x * y
z = matmul(x, y - 0.000002)
z = y
```

Easy multi dimensional arrays (including matrices)

Slices

Strides

Elemental functions

Intrinsics

**Disadvantage: No easy way to parallelize!**

# How to parallelize and offload array notation to a GPU?

```fortran
real, dimension(n, n) :: x, y, z

!$omp target teams distribute parallel do collapse(2)
do i = 1, n
   do j = 1, n
      y(j,i) = sqrt(y(j, i) * x(j, i))
   end do
end do
!$omp target teams distribute parallel do
```

We have to explicitly iterate over the elements and add OpenMP directives…

Disadvantages

- Cumbersome to edit
- Intent not immediately obvious
- Error-prone
  (An intermediate may be needed in some cases)

*omp parallel workshare exists but it can only exploit the `parallel` level (no offloading)

# OpenMP **workdistribute** to the rescue

A new directive in OpenMP 6.0, scheduled to be released later this year.

```
real, dimension(n, n) :: x, y, z

!$omp target teams workdistribute
   y(1:n/2:2,1:n) = 0.1
   y = sqrt(y * x) * x * y
   z = matmul(x, y - 0.000002)
   z = y
!$omp end target teams workdistribute
```

From the OpenMP standard (draft):
Elementals: each element is a work item
Intrinsics: can be parallelized in any way we decide

Parallelization is as *easy as wrapping the code in the directive! (*for the user...)

- Directly nested in (**target**) **teams**
- Supported operations:
  - **Array** and **scalar** assignments
  - Calls to array **intrinsics**, **pure**, and **elemental** functions
- Has to preserve the fortran semantics
  - Preserve statement ordering
  - RHS *appears to finish* execution before assignment to LHS

# RHS *appears to finish* execution before assignment to LHS…?

```
real, dimension(n) :: x

x(1:n) = x(n:1)
```

When (part of) the RHS array overlaps the LHS, element wise assignment will give the wrong result.

The compiler needs to make sure the arrays to not overlap, otherwise, an intermediate is inserted.

# What we want from the implementation

- High level optimizations
  - Merge multiple kernels into a single one (a = a + 2; a = a * 5 -> a = (a + 2) * 5)
  - Removing intermediates (if we can prove they are unnecessary)
  - Intrinsic specific optimizations (e.g. transpose / matmul etc combinations into single calls)
- Ability to use vendor libraries for performance critical operations (e.g. matmul)

# Wait… We need to split the kernel!

Why?

- We need to be able to generate *blas calls from the host in the middle of the original kernel.
- We need to preserve array notation semantics
  (On older GPUs the only way to do a device-wide barrier is to have separate kernels)
- We sometimes need to execute single-threaded code between parallel regions (workdistribute allows scalar code and function calls as well)

*Different targets have different capabilities? The splitting depends on the target?*

**Splitting everything** and **generating all intermediates** is **bad**.

# A small detour:
# OpenMP compilation in Flang

# OpenMP compilation in Flang

## The **host** module

```
module {omp.target = #omp.target<target_cpu = "x86-64">} {

  func.func @func_name(... %args ... ) {
    ...
    %101 = omp.map_info var_ptr(%arg1 : !llvm.ptr, f32) ...
    ...
    omp.target map_entries(%101 -> %arg4, ...) {
    ^bb0(%arg4: !llvm.ptr, ...):
      omp.teams {
        omp.distribute {
          omp.parallel {
            omp.wsloop {
              ...
            }
          }
        }
      }
    }
    ...
  }
}
```

**omp.target** contains the offloading code for the **host**

But we need to do transformation across the host-target boundary…
(and change the interface)

# OpenMP compilation in Flang

## The **host** module

```
module {omp.target = #omp.target<target_cpu = "x86-64">} {

  func.func @func_name(... %args ... ) {
    ...
    %101 = omp.map_info var_ptr(%arg1 : !llvm.ptr, f32) ...
    ...
    omp.target map_entries(%101 -> %arg4, ...) {
    ^bb0(%arg4: !llvm.ptr, ...):
      omp.teams {
        omp.distribute {
          omp.parallel {
            omp.wsloop {
              ...
            }
          }
        }
      }
    }
    ...
  }
}
```

## The **target** modules

```
module {omp.target = #omp.target<target_cpu = "sm_80">} {
```

```
module {omp.target = #omp.target<target_cpu = "gfx801">} {
```

```
module {omp.target = #omp.target<target_cpu = "gfx90a">} {

  func.func @func_name(... %args ... ) {
    ...
    %101 = omp.map_info var_ptr(%arg1 : !llvm.ptr, f32) ...
    ...
    omp.target map_entries(%101 -> %arg4, ...) {
    ^bb0(%arg4: !llvm.ptr, ...):
      omp.teams {
        omp.distribute {
          omp.parallel {
            omp.wsloop {
              ...
            }
          }
        }
      }
    }
    ...
  }
}
```

The host/target interface info is duplicated (function signature, omp.map_info, …)

# Problems

## The frontend decides the host/target interface

And it is the same for **all** targets.

## There is not way to change the host/target interface across modules

Each target is in separate process with independent pipelines

# Single-module OpenMP compilation

```
module {omp.host_container_module} {
  // Forward declaration
  func.func @__omp_outlined_func_name_l400c80(...)
  // Definitions
  omp.module {omp.target = #omp.target<target_cpu = "x86_64">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }
  omp.module {omp.target = #omp.target<target_cpu = "gfx801">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }
  omp.module {omp.target = #omp.target<target_cpu = "gfx90a">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }
  omp.module {omp.target = #omp.target<target_cpu = "sm_80">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }

  func.func @func_name(... %args ... ) {
    ...
    %101 = omp.map_info var_ptr(%arg1 : !llvm.ptr, f32) ...
    ...
    omp.target_call @__omp_outlined_func_name_l400c80 \
         map_entries(%101 -> %arg4, ...)
    ...
  }
}
```

+ Deduplicated host/target interface
+ Allows host/target interface changes

*early proof of concept prototype

# Specialized **omp.target**

```
module {omp.host_container_module} {
  // Forward declaration
  func.func @__omp_outlined_func_name_l400c80(...)
  // Definitions
  omp.module {omp.target = #omp.target<target_cpu = "x86_64">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }
  omp.module {omp.target = #omp.target<target_cpu = "gfx801">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }
  omp.module {omp.target = #omp.target<target_cpu = "gfx90a">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }
  omp.module {omp.target = #omp.target<target_cpu = "sm_80">} {
    func.func @__omp_outlined_func_name_l400c80(...) { ... }
  }

  func.func @func_name(... %args ... ) {
    ...
    %101 = omp.map_info var_ptr(%arg1 : !llvm.ptr, f32) ...
    ...
    omp.target_call @__omp_outlined_func_name_l400c80 \
         map_entries(%101 -> %arg4, ...)
    ...
  }
}
```

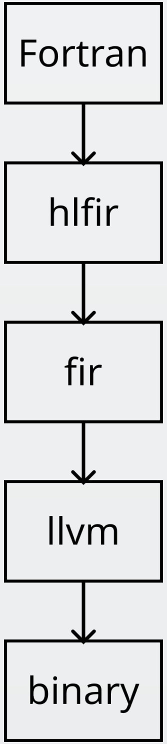
```
module {omp.host_container_module} {
  // Forward declaration
  func.func @__omp_outlined_func_name_l400c80(...)
  ...
  func.func @__omp_specialized_gfx801__omp_outlined_func_name_l400c80(...) {...}
  func.func @__omp_specialized_gfx90a__omp_outlined_func_name_l400c80(...) {...}
  func.func @__omp_specialized_x86_64__omp_outlined_func_name_l400c80(...) {...}
  func.func @__omp_specialized_sm_80__omp_outlined_func_name_l400c80(...) {
    %101 = omp.map_info var_ptr(%arg1 : !llvm.ptr, f32) ...
    ...
    omp.target_specialized map_entries(%101 -> %arg4, ...) {
    ^bb0(%arg4: !llvm.ptr, ...):
      omp.teams {
        omp.distribute {
          omp.parallel {
            omp.wsloop {
              ...
            }
          }
        }
      }
    }
  }

  func.func @func_name(... %args ... ) {
    ...
    omp.call_specialized @__omp_outlined_func_name_l400c80(...)
    ...
  }
}
```

actual sm_80 target code embedded in host

now that we are free to
change the host/target interface **per target**

# Back to workdistribute
# When do we parallelize and split kernels?

# Flang compilation pipeline

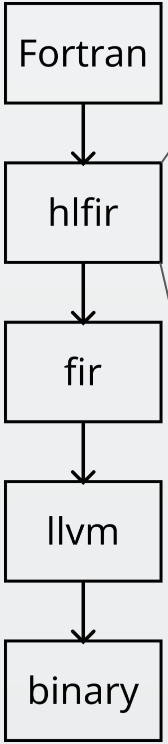


# Flang compilation pipeline

Fortran → hlfir → fir → llvm → binary

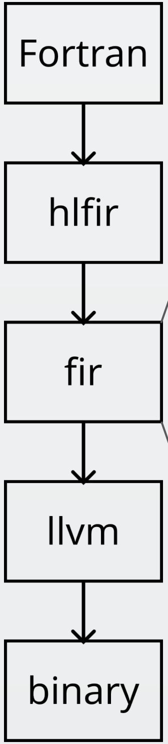
```fortran
real, dimension(n, n) :: x, y, z

y(1:n/2:2,1:n) = 0.1
y = sqrt(y * x) * x * y
z = matmul(x, y - 0.000002)
z = y
```

# Flang compilation pipeline




```
%306 = fir.shape %301, %305 : (index, index) -> !fir.shape<2>
%307 = hlfir.designate %287#0 (%c1_2:%299:%c1_2, %c1_2:%303:%c1_2)  shape %306 : (!fir.box<!fir.array<?x?xf32>>, index, index, index, ind
hlfir.assign %cst_3 to %307 : f32, !fir.box<!fir.array<?x?xf32>>
%308 = hlfir.elemental %286 unordered : (!fir.shape<2>) -> !hlfir.expr<?x?xf32> {
^bb0(%arg14: index, %arg15: index):
  %311 = hlfir.designate %287#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>>, index, index) -> !fir.ref<f32>
  %312 = hlfir.designate %292#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>>, index, index) -> !fir.ref<f32>
  %313 = fir.load %311 : !fir.ref<f32>
  %314 = fir.load %312 : !fir.ref<f32>
  %315 = arith.mulf %313, %314 fastmath<contract> : f32
  %316 = math.sqrt %315 fastmath<contract> : f32
  %317 = hlfir.designate %292#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>>, index, index) -> !fir.ref<f32>
  %318 = fir.load %317 : !fir.ref<f32>
  %319 = arith.mulf %316, %318 fastmath<contract> : f32
  %320 = hlfir.designate %287#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>>, index, index) -> !fir.ref<f32>
  %321 = fir.load %320 : !fir.ref<f32>
  %322 = arith.mulf %319, %321 fastmath<contract> : f32
  hlfir.yield_element %322 : f32
}
hlfir.assign %308 to %287#0 : !hlfir.expr<?x?xf32>, !fir.box<!fir.array<?x?xf32>>
hlfir.destroy %308 : !hlfir.expr<?x?xf32>
%309 = hlfir.elemental %286 unordered : (!fir.shape<2>) -> !hlfir.expr<?x?xf32> {
^bb0(%arg14: index, %arg15: index):
  %311 = hlfir.designate %287#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>>, index, index) -> !fir.ref<f32>
  %312 = fir.load %311 : !fir.ref<f32>
  %313 = arith.subf %312, %cst_0 fastmath<contract> : f32
  hlfir.yield_element %313 : f32
}
%310 = hlfir.matmul %292#0 %309 {fastmath = #arith.fastmath<contract>} : (!fir.box<!fir.array<?x?xf32>>, !hlfir.expr<?x?xf32>) -> !hlfir.
hlfir.assign %310 to %296#0 : !hlfir.expr<?x?xf32>, !fir.box<!fir.array<?x?xf32>>
hlfir.destroy %310 : !hlfir.expr<?x?xf32>
hlfir.destroy %309 : !hlfir.expr<?x?xf32>
hlfir.assign %287#0 to %296#0 : !fir.box<!fir.array<?x?xf32>>, !fir.box<!fir.array<?x?xf32>>
```

# Flang compilation pipeline




```
...
fir.do_loop %arg14 = %c1_8 to %307 step %c1_8 unordered {
  fir.do_loop %arg15 = %c1_8 to %306 step %c1_8 unordered {
    %400 = fir.array_coor %309(%308) %arg15, %arg14 : (!fir.ref<!fir.array<?x?xf32>>, !fir.shape<2>, index, i
    %401 = fir.array_coor %315(%314) %arg15, %arg14 : (!fir.ref<!fir.array<?x?xf32>>, !fir.shape<2>, index, i
    %402 = fir.load %400 : !fir.ref<f32>
    %403 = fir.load %401 : !fir.ref<f32>
    %404 = arith.mulf %402, %403 fastmath<contract> : f32
    %405 = math.sqrt %404 fastmath<contract> : f32
    %406 = arith.mulf %405, %403 fastmath<contract> : f32
    %407 = arith.mulf %406, %402 fastmath<contract> : f32
    %408 = fir.array_coor %335(%308) %arg15, %arg14 : (!fir.heap<!fir.array<?x?xf32>>, !fir.shape<2>, index,
    fir.store %407 to %408 : !fir.ref<f32>
  }
}
%337 = fir.undefined tuple<!fir.box<!fir.array<?x?xf32>>, i1>
...
%348 = fir.call @_FortranAAssign(%345, %346, %347, %c42_i32) : (!fir.ref<!fir.box<none>>, !fir.box<none>, !fi
fir.freemem %349 : !fir.heap<!fir.array<?x?xf32>>
%350 = fir.allocmem !fir.array<?x?xf32>, %306, %307 {bindc_name = ".tmp.array", uniq_name = ""}
...
%369 = fir.call @_FortranAMatmul(%365, %366, %367, %368, %c43_i32) fastmath<contract> : (!fir.ref<!fir.box<no
...
fir.freemem %380 : !fir.heap<!fir.array<?x?xf32>>
%389 = fir.call @_FortranAAssign(%386, %387, %388, %c43_i32_18) : (!fir.ref<!fir.box<none>>, !fir.box<none>,
...
```

# Flang compilation pipeline

Fortran → hlfir → fir → llvm → binary

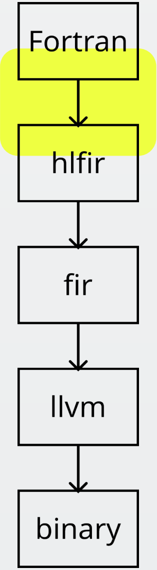
```
^bb4:  // pred: ^bb2
  %132 = llvm.add %107, %21  : i64
  %133 = llvm.sub %108, %21  : i64
  llvm.br ^bb1(%132, %133 : i64, i64)
^bb5(%134: i64, %135: i64):  // 2 preds: ^bb1, ^bb8
  %136 = llvm.icmp "sgt" %135, %19 : i64
  llvm.cond_br %136, ^bb6(%21, %43 : i64, i64), ^bb9
^bb6(%137: i64, %138: i64):  // 2 preds: ^bb5, ^bb7
  %139 = llvm.icmp "sgt" %138, %19 : i64
  llvm.cond_br %139, ^bb7, ^bb8
^bb7:  // pred: ^bb6
  ...
  %166 = llvm.load %152 {tbaa = [#tbaa_tag10]} : !llvm.ptr -> f32
  %167 = llvm.load %165 {tbaa = [#tbaa_tag11]} : !llvm.ptr -> f32
  %168 = llvm.fmul %166, %167  {fastmathflags = #llvm.fastmath<contract>} : f32
  %169 = llvm.intr.sqrt(%168)  {fastmathflags = #llvm.fastmath<contract>} : (f32) -> f32
  %170 = llvm.fmul %169, %167  {fastmathflags = #llvm.fastmath<contract>} : f32
  %171 = llvm.fmul %170, %166  {fastmathflags = #llvm.fastmath<contract>} : f32
  llvm.store %171, %152 {tbaa = [#tbaa_tag10]} : f32, !llvm.ptr
  %172 = llvm.add %137, %21  : i64
  %173 = llvm.sub %138, %21  : i64
  llvm.br ^bb6(%172, %173 : i64, i64)
^bb8:  // pred: ^bb6
  %174 = llvm.add %134, %21  : i64
  %175 = llvm.sub %135, %21  : i64
  llvm.br ^bb5(%174, %175 : i64, i64)
  ...
  %506 = llvm.mul %505, %450  : i64
  %507 = llvm.call @malloc(%506) {bindc_name = ".tmp.array", in_type = !fir.array<?x?xf32>, operandSegmentSizes = array<i32: 0, 2>, uniq_name = ""} : (i64) -> !l
  llvm.br ^bb20(%21, %450 : i64, i64)
^bb20(%508: i64, %509: i64):  // 2 preds: ^bb19, ^bb23
  %510 = llvm.icmp "sgt" %509, %19 : i64
  llvm.cond_br %510, ^bb21(%21, %446 : i64, i64), ^bb24
^bb21(%511: i64, %512: i64):  // 2 preds: ^bb20, ^bb22
  %513 = llvm.icmp "sgt" %512, %19 : i64
  llvm.cond_br %513, ^bb22, ^bb23
^bb22:  // pred: ^bb21
  ...
  %319 = llvm.call @_FortranAMatmul(%29, %15, %13, %318, %18) {fastmathFlags = #llvm.fastmath<contract>} : (!llvm.ptr, !llvm.ptr, !llvm.ptr, !llvm.ptr, i32) -> !
```

# When do we do the parallelization?

- At codegen? (Fortran -> IR)
  - \- We need to generate all intermediates
  - \- We need to split every expression into a separate kernel
  - \- We must make the decision
  - \- Representation is not good for high level optimizations
- At **hlfir** level?
  - \- Non bufferized high level operations (intermediate allocations has not happened yet)
- At **llvm** level?
  - \- We have lost information about loops
  - \- Lowering introduces CFG
  - \+ We are already done with high-level opts
- At **fir** level?
  - \+ Loop information available
  - \+ Arrays are bufferized
  - \+ We are already done with high-level opts

We will parallelize and split kernel at **fir** level

# Introducing **omp.workdistribute**



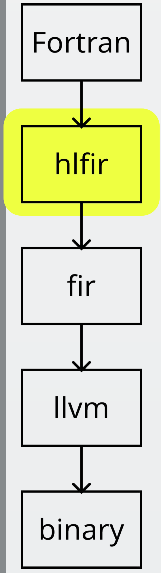

A new MLIR operation in the OpenMP dialect

An intermediate operation to aid in lowering, must disappear (through replacing it with other openmp constructs) in the middle end as there is not lowering to LLVM.

**Region-carrying container operation**
keeps track of the region we need to parallelize through the pipeline

```
omp.target {
  omp.teams {
    omp.workdistribute {
      <mix of temp allocations, array exprs, intrinsics>
    }
  }
}
```

# High level optimizations




Existing **hlfir** optimizations

New optimizations can be added as needed

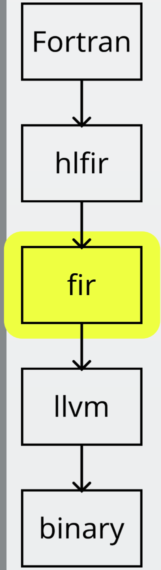
```fortran
real, dimension(n, n) :: x, y, z

y(1:n/2:2,1:n) = 0.1
y = sqrt(y * x) * x * y
z = matmul(x, y - 0.000002)
z = y
```

```
%306 = fir.shape %301, %305 : (index, index) -> !fir.shape<2>
%307 = hlfir.designate %287#0 (%c1_2:%299:%c1_2, %c1_2:%303:%c1_2)  shape %306 :
hlfir.assign %cst_3 to %307 : f32, !fir.box<!fir.array<?x?xf32>>
%308 = hlfir.elemental %286 unordered : (!fir.shape<2>) -> !hlfir.expr<?x?xf32> {
^bb0(%arg14: index, %arg15: index):
  %311 = hlfir.designate %287#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>
  %312 = hlfir.designate %292#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>
  %313 = fir.load %311 : !fir.ref<f32>
  %314 = fir.load %312 : !fir.ref<f32>
  %315 = arith.mulf %313, %314 fastmath<contract> : f32
  %316 = math.sqrt %315 fastmath<contract> : f32
  %317 = hlfir.designate %292#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>
  %318 = fir.load %317 : !fir.ref<f32>
  %319 = arith.mulf %316, %318 fastmath<contract> : f32
  %320 = hlfir.designate %287#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>
  %321 = fir.load %320 : !fir.ref<f32>
  %322 = arith.mulf %319, %321 fastmath<contract> : f32
  hlfir.yield_element %322 : f32
}
hlfir.assign %308 to %287#0 : !hlfir.expr<?x?xf32>, !fir.box<!fir.array<?x?xf32>>
hlfir.destroy %308 : !hlfir.expr<?x?xf32>
%309 = hlfir.elemental %286 unordered : (!fir.shape<2>) -> !hlfir.expr<?x?xf32> {
^bb0(%arg14: index, %arg15: index):
  %311 = hlfir.designate %287#0 (%arg14, %arg15)  : (!fir.box<!fir.array<?x?xf32>
  %312 = fir.load %311 : !fir.ref<f32>
  %313 = arith.subf %312, %cst_0 fastmath<contract> : f32
  hlfir.yield_element %313 : f32
}
%310 = hlfir.matmul %292#0 %309 {fastmath = #arith.fastmath<contract>} : (!fir.bo
hlfir.assign %310 to %296#0 : !hlfir.expr<?x?xf32>, !fir.box<!fir.array<?x?xf32>>
hlfir.destroy %310 : !hlfir.expr<?x?xf32>
hlfir.destroy %309 : !hlfir.expr<?x?xf32>
hlfir.assign %287#0 to %296#0 : !fir.box<!fir.array<?x?xf32>>, !fir.box<!fir.arra
```

# Lowering **omp.workdistribute**: Splitting workdistribute

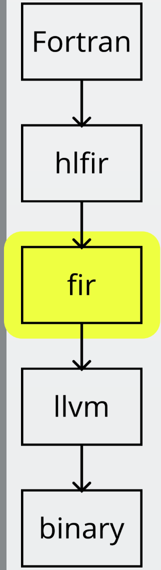
1. Identify parallelizable operations

2. The rest are single-threaded (or contain runtime calls)




```
omp.target {
  omp.teams {
    omp.workdistribute {
     fir.load
     fir.do_loop ... unordered {
       ...
     }
     fir.allocmem ...
     <array descriptor construction>
     matmul(...)
     fir.do_loop ... unordered {
       ...
     }
     <array descriptor construction>
     assign(...)
     fir.freemem ...
    }
  }
}
```

# Lowering **omp.workdistribute**: Splitting workdistribute



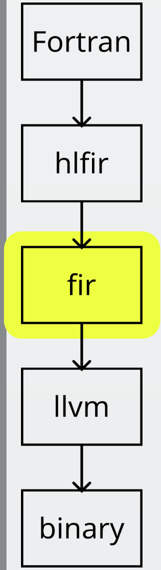

1. Identify parallelizable operations

2. The rest are single-threaded (or contain runtime calls)

3. Fission the **teams**{**workdistribute**} nest

```
omp.target {
  fir.load
  omp.teams {
    omp.workdistribute {
      fir.do_loop ... unordered {
        ...
      }
    }
  }
  fir.allocmem ...
  <array descriptor construction>
  matmul(...)
  omp.teams {
    omp.workdistribute {
      fir.do_loop ... unordered {
        ...
      }
    }
  }
  <array descriptor construction>
  assign(...)
  fir.freemem ...
}
```

# Lowering **omp.workdistribute**: Parallelizing




1. Identify parallelizable operations

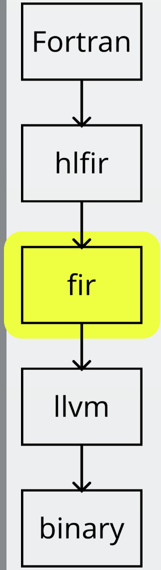
2. The rest are single-threaded (or contain runtime calls)

3. Fission the **teams{workdistribute}** nest

4. Convert **teams{workdistribute}** to **teams{distribute{parallel{}}}** nest

```
omp.target {
  fir.load
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
  fir.allocmem ...
  <array descriptor construction>
  matmul(...)
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
  <array descriptor construction>
  assign(...)
  fir.freemem ...
}
```

# Fissioning **omp.target**




Host / target memory movement is inserted accordingly

```
omp.target {
  fir.load
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
  fir.allocmem ...
  <array descriptor construction>
  matmul(...)
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
  <array descriptor construction>
  assign(...)
  fir.freemem ...
}
```

```
omp.target {
  fir.load
}
omp.target {
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
}
omp_target_alloc(...)
<array descriptor construction>
matmul_omp(...)
omp.target {
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
}
<array descriptor construction>
assign_omp(...)
omp_target_free(...)
```

Now we have valid OpenMP IR
(can be lowered with the existing lowering)

# The Final Piece: Flang OpenMP runtime library

OpenMP versions of the fortran runtime library
(contains array intrinsics like matmul, any_of, transpose, assign, etc)

```
omp.target {
  fir.load
}
omp.target {
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
}
omp_target_alloc(...)
<array descriptor construction>
matmul_omp(...)
omp.target {
  omp.teams {
    omp.distribute {
      omp.parallel {
        omp.wsloop {
          ...
        }
      }
    }
  }
}
<array descriptor construction>
assign_omp(...)
omp_target_free(...)
```

# Memory movement

Thanks to the fortran representation of arrays we know the base pointer and sizes.

We can **automatically** generate (baseline) memory movement

Because we are in the OpenMP ecosystem the user can easily optimize it further if needed

# Evaluation

We have confirmed correctness on hand-coded examples

But we still do not have real application performance evaluation numbers

# Conclusion and Future Work

**Automatic parallelization and offloading of Fortran**

I proposed a set of extensive changes necessary to the OpenMP dialect.

Upstreaming?

OpenMP dialect direction discussion needed

Further **workdistribute** approach discussion