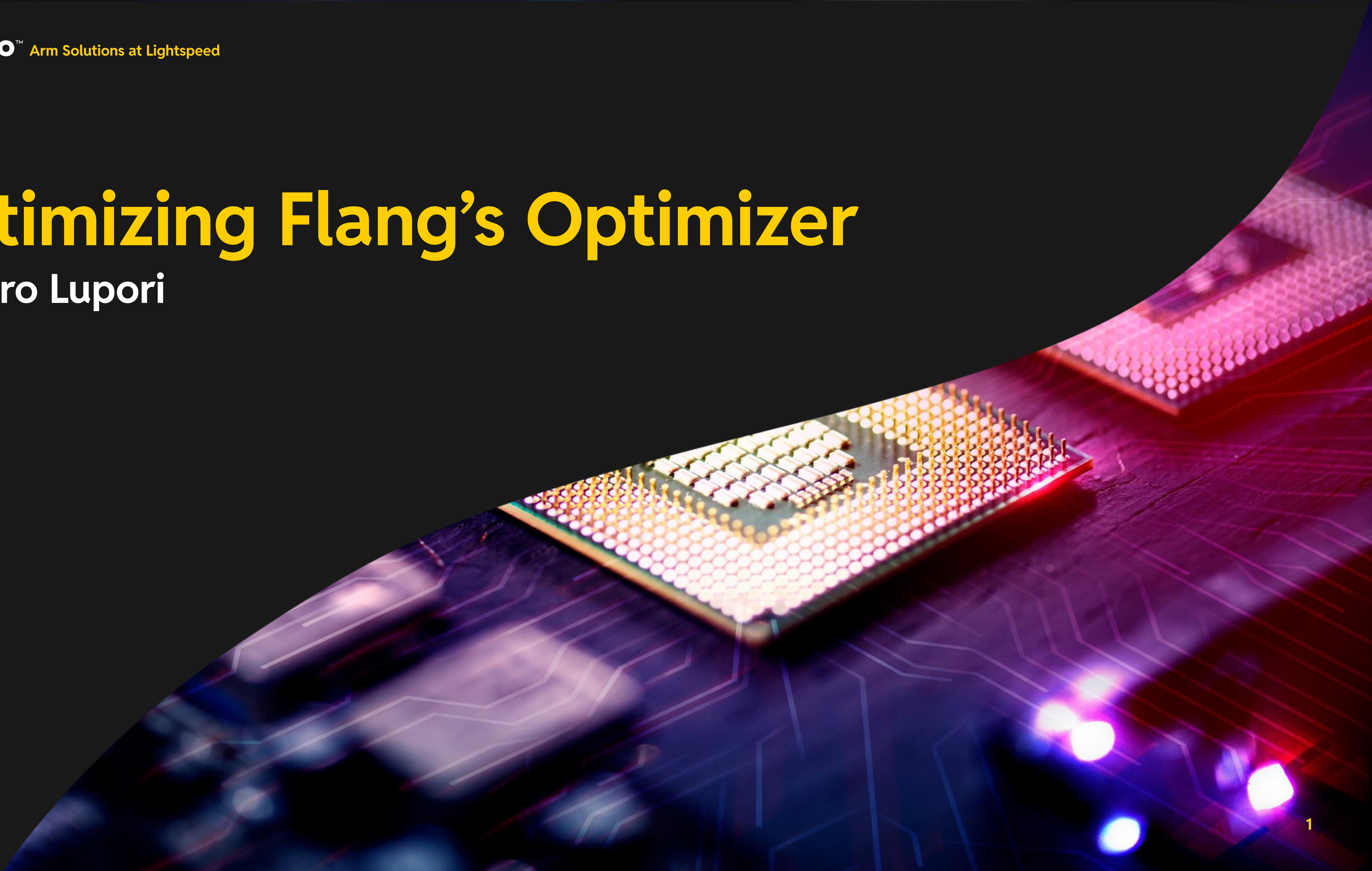
Optimizing Flang's Optimizer

Leandro Lupori



Flang - LLVM's Fortran Front-end

- Flang lowers Fortran to HLFIR (High-Level Fortran IR) + other MLIR dialects

- Flang

```
Fortran  →  HLFIR  →  FIR  →  MLIR
```

- MLIR/LLVM

```
MLIR     →  LLVM IR →  Executable
```

Flang - Fortran to HLFIR

```
function fact(x) result(res)
  integer :: x, res
  if (x <= 1) then
    res = 1
  else
    res = x * fact(x - 1)
  endif
end function
```

```
func.func @_QPfact(%arg0: !fir.ref<i32> {fir.bindc_name = "x"}) -> i32 {
  %1 = fir.alloca i32 {bindc_name = "res", uniq_name = "_QFfactEres"}
  %2:2 = hlfir.declare %1 {uniq_name = "_QFfactEres"} :
    (!fir.ref<i32>) -> (!fir.ref<i32>, !fir.ref<i32>)
  %3:2 = hlfir.declare %arg0 {uniq_name = "_QFfactEx"} :
    (!fir.ref<i32>) -> (!fir.ref<i32>, !fir.ref<i32>)
  %4 = fir.load %3#0 : !fir.ref<i32>
  %c1_i32 = arith.constant 1 : i32
  %5 = arith.cmpi sle, %4, %c1_i32 : i32
  fir.if %5 {
    hlfir.assign %c1_i32 to %2#0 : i32, !fir.ref<i32>
  } else {
    %7 = fir.load %3#0 : !fir.ref<i32>
    %9 = arith.subi %7, %c1_i32 : i32
    %10:3 = hlfir.associate %9 {adapt.valuebyref} :
      (i32) -> (!fir.ref<i32>, !fir.ref<i32>, i1)
    %11 = fir.call @_QPfact(%10#0) fastmath<contract> :
      (!fir.ref<i32>) -> i32
    %12 = arith.muli %7, %11 : i32
    hlfir.assign %12 to %2#0 : i32, !fir.ref<i32>
    hlfir.end_associate %10#1, %10#2 : !fir.ref<i32>, i1
  }
  %6 = fir.load %2#0 : !fir.ref<i32>
  return %6 : i32
}
```

HLFIR/FIR Optimizations

- HLFIR preserves many details of Fortran constructs
  - For instance, variable attributes and array operations are preserved
  - Other details, not useful for optimizations or other transformations are discarded, ex:
    - The distinction between function and subroutine

- Optimizations that rely on Fortran knowledge occur in HLFIR and FIR stages
  - Fortran intrinsinc functions may be replaced with inline code
  - Expressions may be simplified

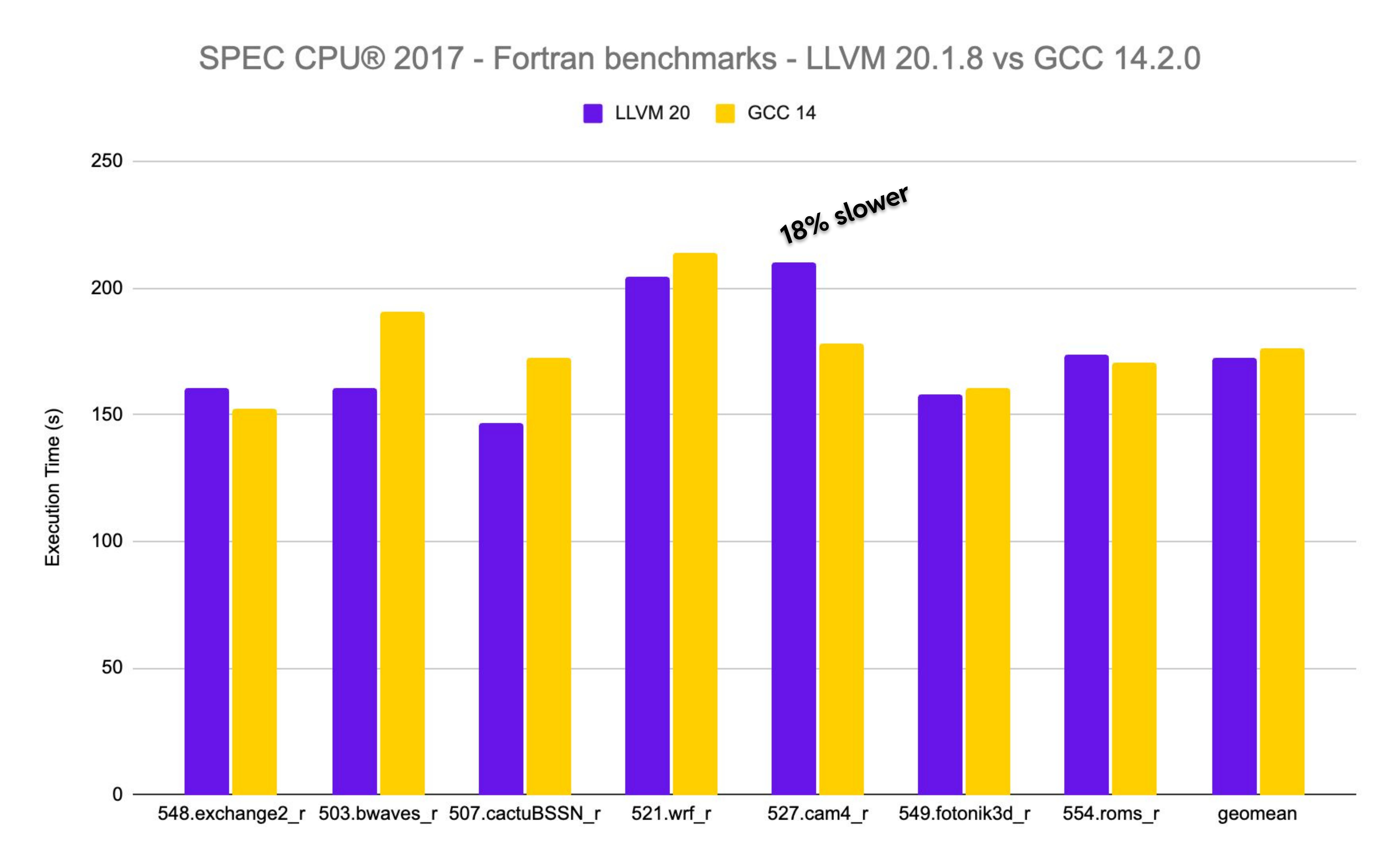
Evaluating Flang Performance

- Setup
  - SPEC CPU® 2017 (rate, 1 copy)
    - Only benchmarks that have Fortran
  - Armv9.0 - Neoverse V2
  - Flags: -O3 -flto

- Goal
  - Flang should be no more than 10% slower than GFortran on any SPEC CPU® 2017 benchmark

Arm Solutions at Lightspeed

LLVM 20 vs GCC 14



Profiling 527.cam4_r

- Perf was used to identify the functions that accounted for the most execution time
  - Ubuntu 24.04 LTS
  - perf record --cpu 1 -e cycles/period=280000000/u
  - Perf ran on cpu 0 and benchmarks on cpu 1, to minimize interference
  - 10 samples per second - CPU frequency 2.8GHz
- The performance issues were spread throughout the program

Profiling 527.cam4_r

perf report -n

- Compare using samples - overhead is relative to execution time, that differs

LLVM 20

Overhead	Samples	Command	Shared Object	Symbol
8.11%	168	cam4_r_base.llv	libm.so.6	[.] log@@GLIBC_2.29
6.90%	143	cam4_r_base.llv	libm.so.6	[.] pow@@GLIBC_2.29
6.42%	133	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMradswPradcswmx
6.23%	129	cam4_r_base.llv	libm.so.6	[.] exp@@GLIBC_2.29
4.93%	102	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMaer_rad_propsPaer_rad_props_sw
3.62%	75	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMstratiformPstratiform_tend
3.43%	71	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMradaePradabs
3.19%	66	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMaer_rad_propsPget_hygro_rad_props
2.80%	58	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMzm_convPientropy
2.32%	48	cam4_r_base.llv	cam4_r_base.llvm20	[.] _QMphysics_typesPphysics_update

GCC 14

Overhead	Samples	Command	Shared Object	Symbol
8.08%	140	cam4_r_base.gcc	libm.so.6	[.] log@@GLIBC_2.29
6.46%	112	cam4_r_base.gcc	libm.so.6	[.] pow@@GLIBC_2.29
5.83%	101	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __radae_MOD_radabs.constprop.0
5.65%	98	cam4_r_base.gcc	libm.so.6	[.] exp@@GLIBC_2.29
4.10%	71	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __radsw_MOD_radcswmx.isra.0
3.87%	67	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __aer_rad_props_MOD_get_hygro_rad_props.isra.0
3.69%	64	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __zm_conv_MOD_ientropy.isra.0
3.29%	57	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __mapz_module_MOD_ppm2m.lto_priv.0
3.12%	54	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __aer_rad_props_MOD_aer_rad_props_sw.constprop.0.isra.0
2.48%	43	cam4_r_base.gcc	cam4_r_base.gcc14	[.] __cldwat_MOD_pcond.constprop.0

Profiling 527.cam4_r

perf annotate

```
       │        0000000000326e80 <_QMaer_rad_propsPaer_rad_props_sw>:
       │          stp     x29, x30, [sp, #-96]!
       │          stp     x28, x27, [sp, #16]
       │          mov     x29, sp
       │          stp     x26, x25, [sp, #32]
       │          stp     x24, x23, [sp, #48]
       │          stp     x22, x21, [sp, #64]
       │          stp     x20, x19, [sp, #80]
       │          sub     sp, sp, #0x12, lsl #12
       │          sub     sp, sp, #0x500
```

LLVM 20

- No debug info…
- Unlike GCC, LLVM sometimes loses debug info when LTO is enabled
- Disabling LTO helps in mapping assembly instructions to Fortran constructs
- Many performance issues occur even with LTO disabled

Linaro Arm Solutions at Lightspeed

Profiling 527.cam4_r

perf annotate - LTO disabled

```
        |        tau      (:,:,:) = -100._r8
        |  398:   stp    x10, x10, [x11]
        |         subs   x9, x9, #0x1
        |         stp    x10, x10, [x11, #16]
        |         stp    x10, x10, [x11, #32]
        |         stp    x10, x10, [x11, #48]
        |         stp    x10, x10, [x11, #64]
        |         stp    x10, x10, [x11, #80]
        |         stp    x10, x10, [x11, #96]
        |         stp    x10, x10, [x11, #112]
```

LLVM 20

```
        |         add    x4, x22, x4
        | 1c0:    mov    x0, x3
        |        tau      (:,:,:) = -100._r8
        | 1c4:    stp    q31, q31, [x0], #32
   3.70 |         cmp    x1, x0
        |       ↑ b.ne   1c4
        |         add    x1, x1, #0x360
        |         add    x3, x3, #0x360
        |         cmp    x4, x1
        |       ↑ b.ne   1c0
```

GCC 14

Profiling 527.cam4_r

perf annotate - LTO disabled

- The LTO version used ***stp q0, q0, [x11, #<offs>]*** instead (**X** registers are 64-bit wide while **Q** registers are 128-bit)

- Both versions used many ***stp*** instructions and some ***str*** instructions

- LLVM didn’t seem to realize that a single loop could be used to initialize the 3-dimensional array ***tau***

Digging in

- To make analysis easier a test program with a non-optimizable assignment to a 3D array (in a loop) was used

HLFIR:

```
func.func private @_QFPinit_tau() attributes {fir.host_symbol = @_QQmain, llvm.linkage = #llvm.linkage<internal>} {
  ...
  %10 = fir.shape %c4, %c27, %c19 : (index, index, index) -> !fir.shape<3>
  %11:2 = hlfir.declare %9(%10) {uniq_name = "_QFEtau"} : (!fir.ref<!fir.array<4x27x19xf64>>, !fir.shape<3>) ->
    (!fir.ref<!fir.array<4x27x19xf64>>, !fir.ref<!fir.array<4x27x19xf64>>)
  %cst = arith.constant -1.000000e+02 : f64
  hlfir.assign %cst to %11#0 : f64, !fir.ref<!fir.array<4x27x19xf64>>
  return
}
```

Digging in

FIR:

```
func.func private @_QFPinit_tau() attributes {fir.host_symbol = @_QQmain, llvm.linkage = #llvm.linkage<internal>} {
  ...
  %9 = fir.shape %c4, %c27, %c19 : (index, index, index) -> !fir.shape<3>
  %10 = fir.declare %8(%9) {uniq_name = "_QFEtau"} : (!fir.ref<!fir.array<4x27x19xf64>>, !fir.shape<3>) ->
    !fir.ref<!fir.array<4x27x19xf64>>
  fir.do_loop %arg0 = %c1 to %c19 step %c1 unordered {
    fir.do_loop %arg1 = %c1 to %c27 step %c1 unordered {
      fir.do_loop %arg2 = %c1 to %c4 step %c1 unordered {
        %11 = fir.array_coor %10(%9) %arg2, %arg1, %arg0 :
          (!fir.ref<!fir.array<4x27x19xf64>>, !fir.shape<3>, index, index, index) -> !fir.ref<f64>
        fir.store %cst to %11 : !fir.ref<f64>
      }
    }
  }
  return
}
```

Digging in

LLVM:

```
.preheader2.i:                                    ; preds = %.preheader2.i.preheader, %.preheader2.i
  %33 = phi i64 [ %34, %.preheader2.i ], [ 1, %.preheader2.i.preheader ]
  %.idx.i = mul nuw nsw i64 %33, 864
  %gep5.i = getelementptr i8, ptr getelementptr (i8, ptr @_QFEtau, i64 -904), i64 %.idx.i
  %gep.i = getelementptr i8, ptr %gep5.i, i64 40
  store double -1.000000e+02, ptr %gep.i, align 16, !tbaa !8
  %gep.1.i = getelementptr i8, ptr %gep5.i, i64 48
  store double -1.000000e+02, ptr %gep.1.i, align 8, !tbaa !8
  %gep.2.i = getelementptr i8, ptr %gep5.i, i64 56
  store double -1.000000e+02, ptr %gep.2.i, align 16, !tbaa !8
  ...
```

- The 2 inner loops were completely unrolled
- But this is a contiguous array
- Maybe LLVM can produce a more optimized IR if Flang lowers multidimensional array assignments using a single loop?

Multidimensional array linearization

- [flang] Optimize assignments of multidimensional arrays #146408

- flang/lib/Optimizer/HLFIR/Transforms/OptimizedBufferization.cpp

```
llvm::LogicalResult BroadcastAssignBufferization::matchAndRewrite(
    hlfir::AssignOp assign, mlir::PatternRewriter &rewriter) const {
  ...
  if (lhs.isSimplyContiguous() && extents.size() > 1) {
    // Flatten the array to use a single assign loop, that can be better optimized.
    ...
    hlfir::LoopNest loopNest =
        hlfir::genLoopNest(loc, builder, flatExtents, /*isUnordered=*/true,
                           flangomp::shouldUseWorkshareLowering(assign));
    builder.setInsertionPointToStart(loopNest.body);
    mlir::Value arrayElement =
        builder.create<hlfir::DesignateOp>(loc, fir::ReferenceType::get(eleTy),
                                           flatArray, loopNest.oneBasedIndices);
    builder.create<hlfir::AssignOp>(loc, rhs, arrayElement);
  }
  ...
}
```

Multidimensional array linearization

FIR:

```
func.func private @_QFPinit_tau() attributes {fir.host_symbol = @_QQmain, llvm.linkage = #llvm.linkage<internal>} {
  ...
  %10 = fir.shape %c4, %c27, %c19 : (index, index, index) -> !fir.shape<3>
  %11 = fir.declare %9(%10) {uniq_name = "_QFEtau"} : (!fir.ref<!fir.array<4x27x19xf64>>, !fir.shape<3>) ->
    !fir.ref<!fir.array<4x27x19xf64>>
  %12 = fir.convert %11 : (!fir.ref<!fir.array<4x27x19xf64>>) -> !fir.ref<!fir.array<2052xf64>>
  fir.do_loop %arg0 = %c1 to %c2052 step %c1 unordered {
    %c2052_0 = arith.constant 2052 : index
    %13 = fir.shape %c2052_0 : (index) -> !fir.shape<1>
    %14 = fir.array_coor %12(%13) %arg0 : (!fir.ref<!fir.array<2052xf64>>, !fir.shape<1>, index) -> !fir.ref<f64>
    fir.store %cst to %14 : !fir.ref<f64>
  }
  return
}
```

Multidimensional array linearization

LLVM:

```
vector.ph:                                        ; preds = %27, %_QFPinit_tau.exit
  %32 = phi i64 [ %36, %_QFPinit_tau.exit ], [ %30, %27 ]
  br label %vector.body, !dbg !69

vector.body:                                      ; preds = %vector.body, %vector.ph
  %index = phi i64 [ 0, %vector.ph ], [ %index.next, %vector.body ]
  %33 = getelementptr double, ptr @_QFEtau, i64 %index, !dbg !69
  %34 = getelementptr i8, ptr %33, i64 16, !dbg !69
  store <2 x double> splat (double -1.000000e+02), ptr %33, align 16, !dbg !69, !tbaa !73
  store <2 x double> splat (double -1.000000e+02), ptr %34, align 16, !dbg !69, !tbaa !73
  %index.next = add nuw i64 %index, 4
  %35 = icmp eq i64 %index.next, 2052, !dbg !69
  br i1 %35, label %_QFPinit_tau.exit, label %vector.body, !dbg !69, !llvm.loop !80

_QFPinit_tau.exit:                                ; preds = %vector.body
  %36 = add nsw i64 %32, -1, !dbg !68
  %37 = icmp sgt i64 %32, 1, !dbg !68
  br i1 %37, label %vector.ph, label %._crit_edge, !dbg !68
```

Multidimensional array linearization

AArch64:

```
  mov x8, #-4586634745500139520
  mov x9, x0
  dup v0.2d, x8
  adrp  x8, _QFEtau+16
  add x8, x8, :lo12:_QFEtau+16
  mov x10, x8
  mov w11, #2052
.LBB0_7:
  subs  x11, x11, #4
  stp q0, q0, [x10, #-16]
  add x10, x10, #32
  b.ne  .LBB0_7
```

- Test program

```
     | 328:   mov     x8, #0xc059000000000000        // #-4586634745500139520
     |        mov     w9, #0x804                     // #2052
     |        dup     v0.2d, x8
     |        add     x8, x20, #0x10
     | 338:   subs    x9, x9, #0x4
     |        stp     q0, q0, [x8, #-16]
     |        add     x8, x8, #0x20
     |        b.ne    338
```

- 527.cam4_r

Results

Multidimensional array linearization

- In the end, there was practically no difference in 527.cam4_r
    - aer_rad_props_sw is a large function
    - It spent less than 10% of its time in array initialization
    - There were other performance issues in aer_rad_props_sw

Results

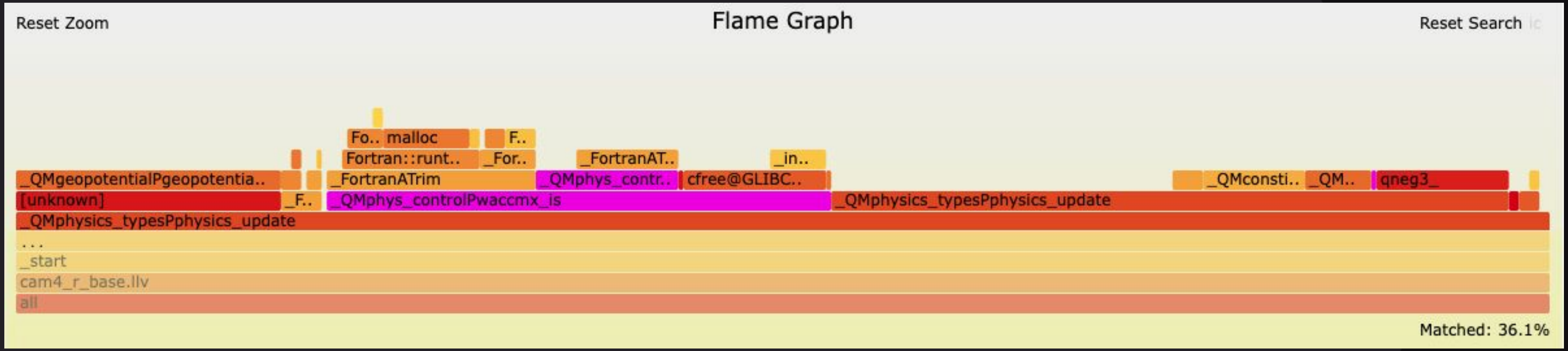
Multidimensional array linearization

- However, there was a substantial performance improvement in other test programs
- The number of instructions per array initialization was reduced from 38 to 8

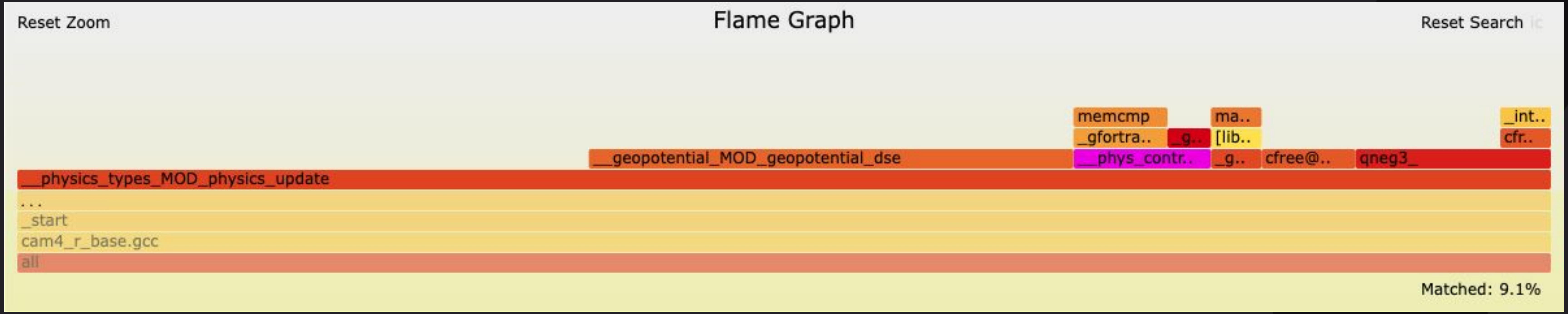
Array size	Speedup
16KB	8%
64KB	28%
128KB	31%
256KB	32%
512KB+	0%

Profiling 527.cam4_r (again)

Using flame graphs to find performance differences



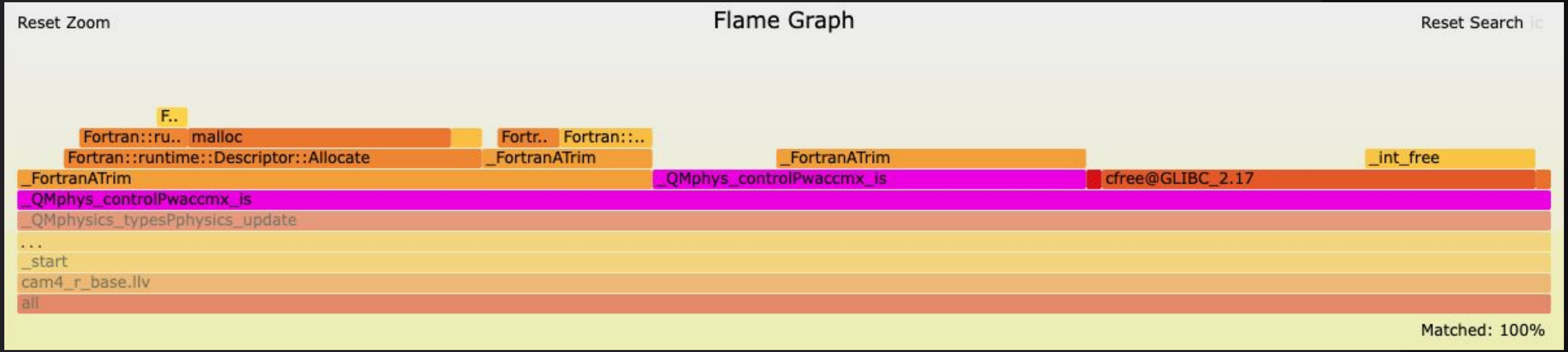
LLVM



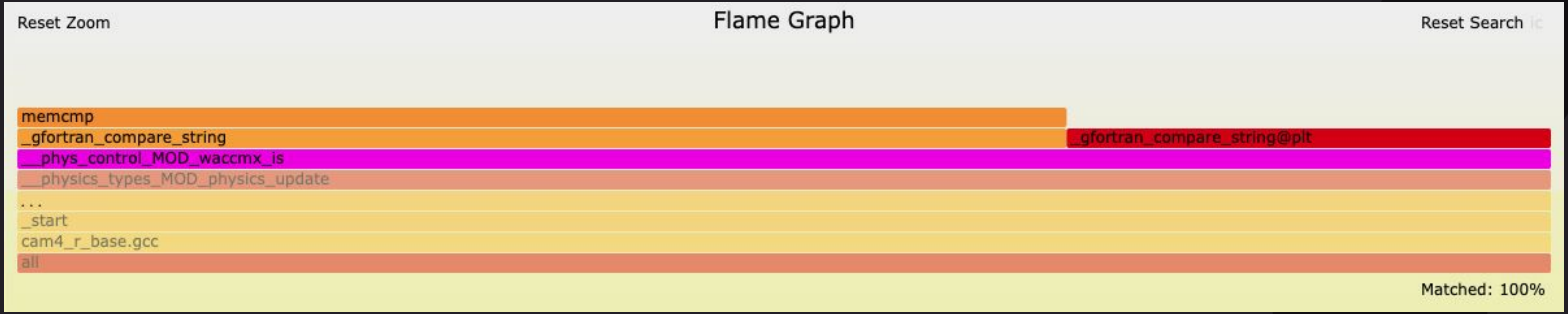
GCC 15

Profiling 527.cam4_r (again)

waccmx_is



LLVM



GCC 15

linaro™ Arm Solutions at Lightspeed

Profiling 527.cam4_r (again)

- waccmx_is = trim(name) == trim(waccmx_opt)

```
0.90 │     → bl    _FortranATrim
     │       ...
0.90 │     → bl    _FortranATrim
     │       ...
9.91 │       cmp   w12, w13
     │       cset  w12, hi // hi = pmore
     │       ...
0.90 │     → bl    free@plt
     │       ...
     │     → bl    free@plt
```

LLVM

```
     │    waccmx_is = (trim(name) == trim(waccmx_opt))
     │      add  x3, x3, #0x80
     │      mov  x1, x2
     │      mov  x2, #0x10                        // #16
     │    → bl   _gfortran_compare_string@plt
     │      cmp  w0, #0x0
     │    end function waccmx_is
     │      cset w0, eq  // eq = none
     │      ldp  x29, x30, [sp], #16
     │    ← ret
```

GCC 15

Fortran relational operations

- Fortran character strings with fixed length are padded with blanks on the right

- Fortran 2018 language specification
  10.1.5.5.1 Interpretation of relational intrinsic operations
  “For a character relational intrinsic operation, the operands are compared one character at a time in order, beginning with the first character of each character operand. ***If the operands are of unequal length, the shorter operand is treated as if it were extended on the right with blanks to the length of the longer operand****.”*

- Conclusion: the calls to trim() are unnecessary and can be removed

HLFIR Expression Simplification

- The first step was to add the hlfir.char_trim operation
  - This made it easier to identify and remove calls to trim()
  - When removal was not possible the operation was converted to a runtime call

- With it a pass to simplify the comparison of characters was written

```
trim(x) == trim(y) → x == y
```

- This optimization shouldn't cause performance regressions
  - It is unlikely to preclude others which bring more significant gains

HLFIR Expression Simplification

- flang/lib/Optimizer/HLFIR/Transforms/ExpressionSimplification.cpp

```
llvm::LogicalResult
matchAndRewrite(hlfir::CharTrimOp trimOp,
                mlir::PatternRewriter &rewriter) const override {
  int trimUses = std::distance(trimOp->use_begin(), trimOp->use_end());
  auto cmpCharOp = getFirstUser<hlfir::CmpCharOp>(trimOp);
  auto destroyOp = getLastUser<hlfir::DestroyOp>(trimOp);
  if (!cmpCharOp || !destroyOp || trimUses != 2)
    return rewriter.notifyMatchFailure(
        trimOp, "hlfir.char_trim is not used (only) by hlfir.cmpchar");
  rewriter.eraseOp(destroyOp);
  rewriter.replaceOp(trimOp, trimOp.getChr());
  return mlir::success();
}
```

HLFIR Expression Simplification

HLFIR:

```
%7 = hlfir.char_trim %4#0 : (!fir.boxchar<1>) -> !hlfir.expr<!fir.char<1,?>>
%8 = hlfir.char_trim %6#0 : (!fir.boxchar<1>) -> !hlfir.expr<!fir.char<1,?>>
%9 = hlfir.cmpchar eq %7 %8 : (!hlfir.expr<!fir.char<1,?>>, !hlfir.expr<!fir.char<1,?>>) -> i1
%10 = fir.convert %9 : (i1) -> !fir.logical<4>
hlfir.assign %10 to %2#0 : !fir.logical<4>, !fir.ref<!fir.logical<4>>
hlfir.destroy %8 : !hlfir.expr<!fir.char<1,?>>
hlfir.destroy %7 : !hlfir.expr<!fir.char<1,?>>
```

Optimized HLFIR:

```
%7 = hlfir.cmpchar eq %4#0 %6#0 : (!fir.boxchar<1>, !fir.boxchar<1>) -> i1
%8 = fir.convert %7 : (i1) -> !fir.logical<4>
hlfir.assign %8 to %2#0 : !fir.logical<4>, !fir.ref<!fir.logical<4>>
```

Results

HLFIR Expression Simplification

- 527.cam4_r time went from 217 to 208.8 seconds
- Estimated 3.5% speedup
  (considering margin of error and variability)

Function Inlining

- Challenge: different compilers inline different functions
- Considering only the time spent on *radcswmx*, it looks like GCC is twice as slow
- However, a call graph or flame graph shows this is mostly due to *raddedmx* being inlined by GCC but not by LLVM

```
  Children      Self  Command          Shared Object           Symbol
-    9.52%     3.23%  cam4_r_base.llv  cam4_r_base.llvm_trim0  [.] _QMradswPradcswmx
   - 6.30% _QMradswPradcswmx
      + 3.25% _QMradswPraddedmx
        0.91% _FortranAAssign
        0.61% __memset_zva64
   + 3.23% _start
```

LLVM

```
  Children      Self  Command          Shared Object           Symbol
-   11.28%     6.73%  cam4_r_base.gcc  cam4_r_base.gcc15_o3_g  [.] __radsw_MOD_radcswmx
   + 6.73% _start
   - 4.55% __radsw_MOD_radcswmx
        3.14% exp@@GLIBC_2.29
        0.51% __memset_zva64
```

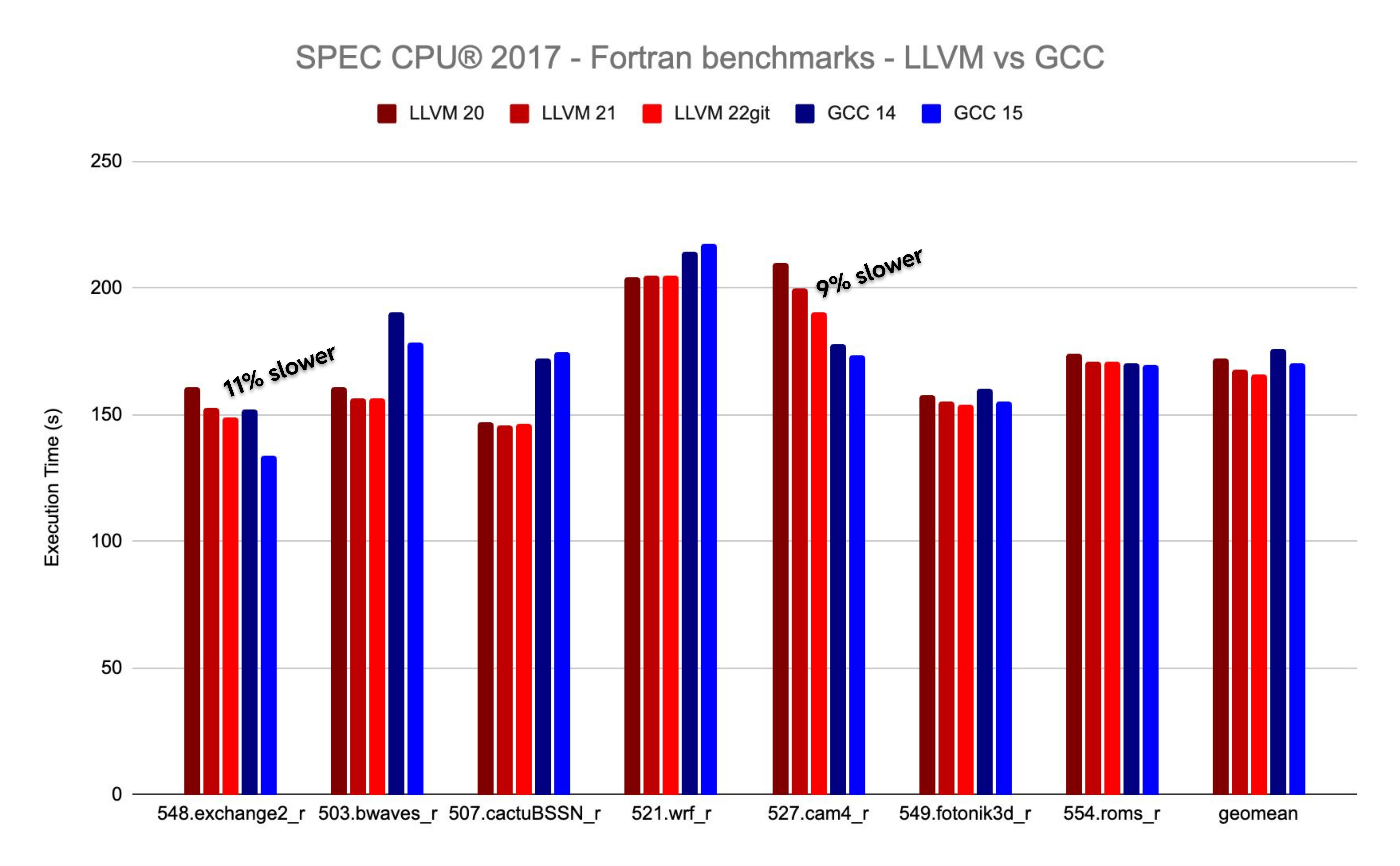
GCC 15

Community Contributions

- Other developers also contributed with Flang performance enhancements
- Considering single thread performance only, these are some recent examples:
  - [flang] Simplify hlfir.index in a few limited cases
  - [flang][driver] Accelerate complex division when -ffast-math is specified
  - [flang] Create TBAA subtree for COMMON block variables
  - [flang] Lower hlfir.cmpchar into inline implementation in simplify-hlfir-intrinsics
  - [flang] Inline hlfir.eoshift during HLFIR intrinsics simplication
  - [flang][runtime] Optimize Descriptor::FixedStride()
  - [flang][runtime] Speed up initialization & destruction
  - [flang] Optimize redundant array repacking

Arm Solutions at Lightspeed

LLVM vs GCC



Future Work

- Improve the performance of 548.exchange2_r
- Check Flang performance on SPEC CPU® v8
- Test other compiler flags: -fno-lto, -mcpu=neoverse-v2
- Check Flang OpenMP performance with SPEC speed

linaro.org

Thank you

leandro.lupori@linaro.org