A walk through *Flang* OpenMP lowering: From FIR to LLVMIR

Arnamoy Bhattacharyya*, Peixin Qiao, Bryan Chan

Huawei Technologies Canada

Presented in LLVM Workshop, within CGO 2022, Apr‘22

Why this talk?

- LLVM Flang (replacing Classic Flang) under active development.
  - Written in C++17
  - Uses MLIR
- Volunteers needed for contribution in OpenMP
- Parsing support is there for OpenMP 4.5
- Significant portion of sema checks are done
  - OpenMP 1.1 support VERY soon
- OpenMP 2.5, 3.1 etc are needing active development.
- A “getting started” for lowering OpenMP code for LLVM Flang

*https://docs.google.com/spreadsheets/d/1FvHPuSkGbl4mQZRAwCIndvQx9dQboffiD-xD0oqxgU0/edit#gid=0

HUAWEI

Goal: Implement the lowering of basic SIMD construct

From OpenMP5.0 standard Section 2.9.3.1

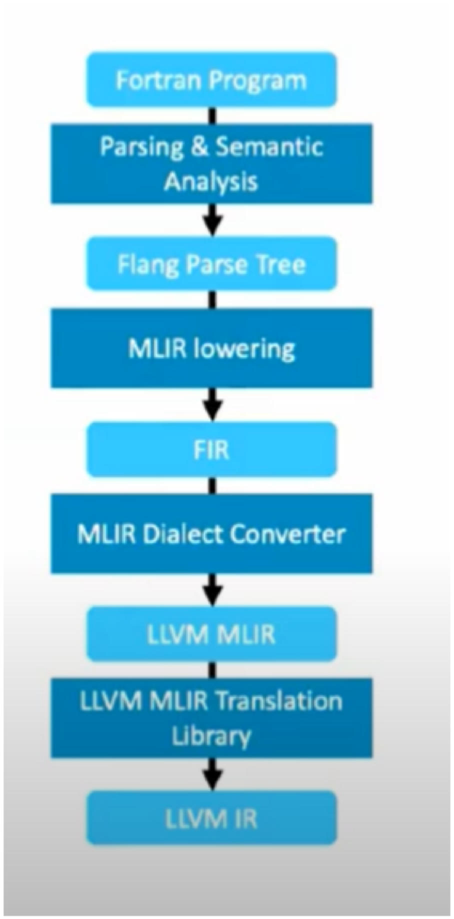
Summary The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

`!$omp do simd` ~~[clause[[,] clause] ...]~~
    do-loops
[**`!$omp end do simd`** ~~[**nowait**]]~~

*https://www.openmp.org/spec-html/5.0/openmpsu42.html

Flang compiler flow

- Parses Fortran 2018
- Performs Semantic Checks
- Lowers to high level IR **FIR**
  - LLVM IR is too low level for Fortran
  - Uses the MLIR framework
- Converts to a lower level IR, LLVM MLIR
- Lowers to LLVM IR



*

* Picture courtesy: Kiran Chandramohan, ARM

HUAWEI

Background MLIR

- Multi-level Intermediate Representation
- A new approach for building compiler infrastructure
  - Can use to build SSA-based IR
  - Provides a declarative system for defining IRs
  - Provides common infrastructure (printing, parsing, location tracking, pass management etc.)
- Flang compiler uses MLIR based FIR dialect as its IR
- FIR models the Fortran language portion
  - **Does not** have a representation for OpenMP constructs
- Add a dialect in MLIR for OpenMP
  - MLIR provides common framework for representing OpenMP and Fortran
  - Makes OpenMP codegen reuseable

HUAWEI

OpenMP IRBuilder

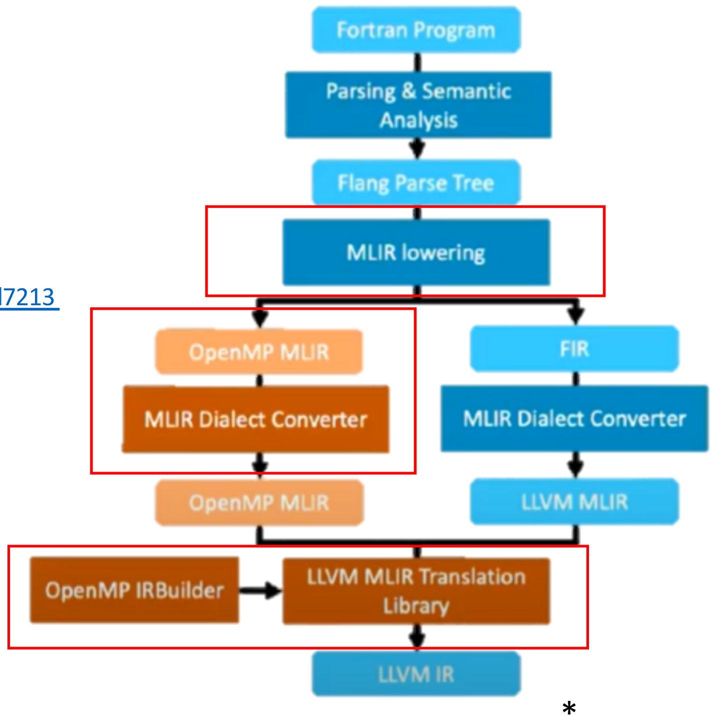
- Generating LLVM IR involves two important tasks
  - Inserting calls to OpenMP runtime
  - Outlining OpenMP regions
- Code exists in clang for these tasks
  - Reuse codegen from Clang
- Refactor codegen for OpenMP constructs in Clang and move to LLVM directory
  - `llvm/lib/Frontend/OpenMP`

HUAWEI

OpenMP plan for Flang

MLIR Support →
https://reviews.llvm.org/rG0e9198c3e95adced7213999dcd14daed4acfd16c

OMPIRBuilder support →
https://reviews.llvm.org/rG9fbd33ad623d2b576fc563545bbdf2c257cdf709



* Picture courtesy: Kiran Chandramohan, ARM

Implementation of lowering of SIMD construct

Steps for implementation

1. Read about the behavior of the construct/clause from OpenMP website (refer OpenMP spec for details)

About SIMD construct

2.9.3.1 `simd` **Construct**

Summary The `simd` construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

```
!$omp simd
   do-loops
[!$omp end simd]
```

HUAWEI

Steps for implementation

1. Read about the behavior of the construct/clause from OpenMP website
2. Identify the IR changes necessary

Visualize the changes necessary in the final IR

- Write a simple test case and look at the IR

```
void omp_simd() {
  int i = 0;
  int a[16];
  #pragma omp simd
  for (int i=0; i <16; i++) {
      a[i] = i;
  }
  return;
}
```

HUAWEI

Visualize the changes necessary in the final IR

```
7:                                                ; preds = %4
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sext i32 %9 to i64
  %11 = getelementptr inbounds [16 x i32], [16 x i32]* %2, i64 0, i64 %10
  store i32 %8, i32* %11, align 4
  br label %12

12:                                               ; preds = %7
  %13 = load i32, i32* %3, align 4
  %14 = add nsw i32 %13, 1
  store i32 %14, i32* %3, align 4
  br label %4, !llvm.loop !5

15:                                               ; preds = %4

  ret void
}

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "n
o-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+cx8,+
fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"openmp", i32 50}
!2 = !{i32 7, !"uwtable", i32 2}
!3 = !{i32 7, !"frame-pointer", i32 2}
!4 = !{!"clang version 15.0.0 (https://github.com/llvm/llvm-project.git 7764a05d9c20b1340fa3dbbdf3ac99743
b6deccf)"}
!5 = distinct !{!5, !6}
!6 = !{!"llvm.loop.mustprogress"}
```

```
9:                                                ; preds = %6
  %10 = load i32, i32* %4, align 4, !llvm.access.group !5
  %11 = mul nsw i32 %10, 1
  %12 = add nsw i32 0, %11
  store i32 %12, i32* %5, align 4, !llvm.access.group !5
  %13 = load i32, i32* %5, align 4, !llvm.access.group !5
  %14 = load i32, i32* %5, align 4, !llvm.access.group !5
  %15 = sext i32 %14 to i64
  %16 = getelementptr inbounds [16 x i32], [16 x i32]* %2, i64 0, i64 %15
  store i32 %13, i32* %16, align 4, !llvm.access.group !5
  br label %17

17:                                               ; preds = %9
  br label %18

18:                                               ; preds = %17
  %19 = load i32, i32* %4, align 4, !llvm.access.group !5
  %20 = add nsw i32 %19, 1
  store i32 %20, i32* %4, align 4, !llvm.access.group !5
  br label %6, !llvm.loop !6

21:                                               ; preds = %6
  store i32 16, i32* %5, align 4
  ret void
}

attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-leg
o-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "t
fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3}
!llvm.ident = !{!4}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"openmp", i32 50}
!2 = !{i32 7, !"uwtable", i32 2}
!3 = !{i32 7, !"frame-pointer", i32 2}
!4 = !{!"clang version 15.0.0 (https://github.com/llvm/llvm-pr 7764a05d9c
b6deccf)"}
!5 = distinct !{}
!6 = distinct !{!6, !7, !8}
!7 = !{!"llvm.loop.parallel_accesses", !5}
!8 = !{!"llvm.loop.vectorize.enable", i1 true}
```

Summary of IR changes

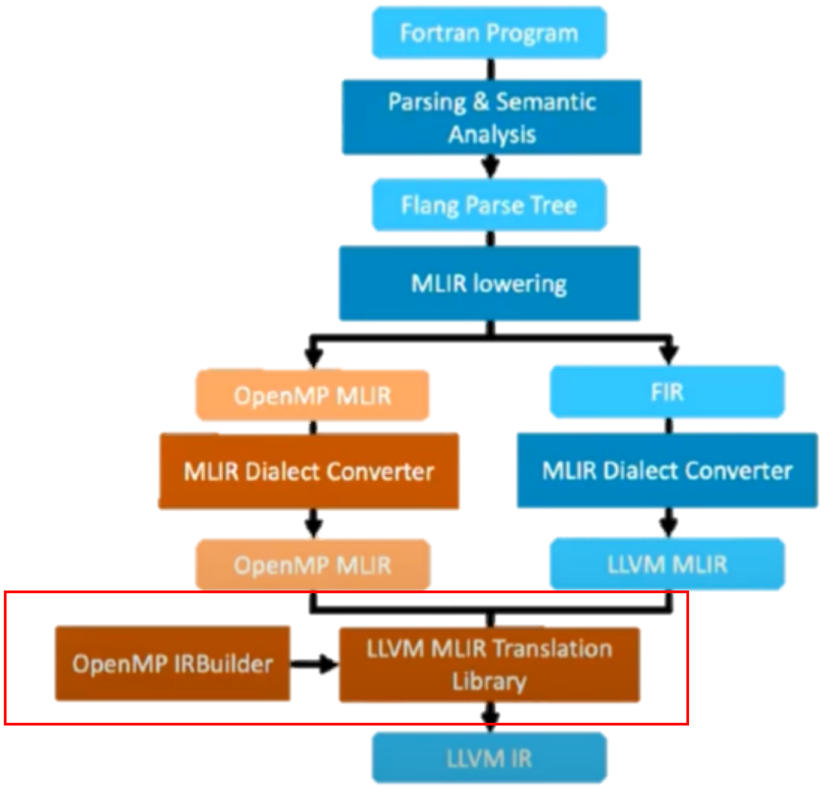
- Insert llvm.access.group metadata to the Memory access instructions in the loop
- Change the llvm.loop metadata associated with the loop
- **No need** to insert any omp runtime calls

HUAWEI

Steps for implementation

1. Read about the behavior of the construct/clause from OpenMP website
2. Identify the IR changes necessary
3. Identify if IRBuilder support is needed, implement

Where is IRBuilder used?



Is IRBuilder support needed?

- Rule of thumb:
  - If implementing lowering of new **directives**, the answer is most probably **yes**
  - For implementing **clauses**, the answer is probably **no**

```
!$omp do simd
   do-loops
[!$omp end do simd
```

Directive → yes

```
!$omp do simd lastprivate(a)
   do-loops
[!$omp end do simd
```

Clause → No

HUAWEI

IRBuilder support for SIMD

- Steps necessary to support Parse tree -> LLVM IR lowering*
  - Create a loop
  - Add metadata
- OMPIRBuilder has an existing struct to represent canonical loop and an API to create one.

```
class CanonicalLoopInfo {
  friend class OpenMPIRBuilder;

private:
  BasicBlock *Header = nullptr;
  BasicBlock *Cond = nullptr;
  BasicBlock *Latch = nullptr;
  BasicBlock *Exit = nullptr;

  /// Add the control blocks of this loop to \p BBs.
  ///
  /// This does not include any block from the body, i
  /// by getBody().
  ///
```

```
CanonicalLoopInfo *
OpenMPIRBuilder::createCanonicalLoop(const LocationDescription &Loc,
                                     LoopBodyGenCallbackTy BodyGenCB,
                                     Value *TripCount, const Twine &Nam
  BasicBlock *BB = Loc.IP.getBlock();
  BasicBlock *NextBB = BB->getNextNode();

  CanonicalLoopInfo *CL = createLoopSkeleton(Loc.DL, TripCount, BB->get
                                             NextBB, NextBB, Name);
  BasicBlock *After = CL->getAfter();

  // If location is not set, don't connect the loop.
  if (updateToLocation(Loc)) {
```

* for Clang

HUAWEI

Strategy for IRBuilder support

- When we encounter SIMD directive in the parse tree, create a canonical loop CL first using the API
  - In `clang/lib/CodeGen/CGStmt.cpp`
- Define a function that can take the newly created CL and apply the metadata changes necessary.

```
void OpenMPIRBuilder::applySimd(CanonicalLoopInfo *CL)
```

  - In `llvm/lib/Frontend/OpenMP/OMPIRBuilder.cpp`

HUAWEI

1. Creating the canonical loop

- `clang/lib/CodeGen/CGStmt.cpp` → code to emit LLVM code from AST Stmt nodes

```
void CodeGenFunction::EmitStmt(const Stmt *S, ArrayRef<const Attr *> Attrs) {
…..
switch (S->getStmtClass()) {

…
  case Stmt::OMPSimdDirectiveClass:
      EmitOMPSimdDirective(cast<OMPSimdDirective>(*S));
      break;
```

1. Creating the canonical loop

```cpp
void CodeGenFunction::EmitOMPSimdDirective(const OMPSimdDirective &S) {
  bool UseOMPIRBuilder =
      CGM.getLangOpts().OpenMPIRBuilder && isSupportedByOpenMPIRBuilder(S);
  if (UseOMPIRBuilder) {
    auto &&CodeGenIRBuilder = [this, &S](CodeGenFunction &CGF,
                                         PrePostActionTy &) {
      // Emit the associated statement and get its loop representation.
      const Stmt *Inner = S.getRawStmt();
      llvm::CanonicalLoopInfo *CLI =
          EmitOMPCollapsedCanonicalLoopNest(Inner, 1);

      llvm::OpenMPIRBuilder &OMPBuilder =
          CGM.getOpenMPRuntime().getOMPBuilder();
      // Add SIMD specific metadata
      OMPBuilder.applySimd(CLI);

      return;
    };
    {
      CGM.getOpenMPRuntime().emitInlinedDirective(*this, OMPD_simd,
                                                  CodeGenIRBuilder);
    }
    return;
  }
```

Uses createCanonicalLoop()

Function that is called while lowering the SIMD directive

Check is compiler is using OMPIRBuilder, also check for any condition e.g. unsupported clauses

Lowering code

Lambda call

CGM → per module state

HUAWEI

2. Attaching the metadata (applySimd())

- Getting the `llvm::Loop` from the `CanonicalLoopInfo` struct
  - A bit hacky currently.
- Extracting the Basic blocks which needs to be modified with new metadata
- Find memref instructions in the BasicBlocks and attach metadata.

HUAWEI

2. Attaching the metadata

```
void OpenMPIRBuilder::applySimd(DebugLoc, CanonicalLoopInfo *CanonicalLoop) {
  LLVMContext &Ctx = Builder.getContext();

  Function *F = CanonicalLoop->getFunction();

  FunctionAnalysisManager FAM;
  FAM.registerPass([]() { return DominatorTreeAnalysis(); });
  FAM.registerPass([]() { return LoopAnalysis(); });
  FAM.registerPass([]() { return PassInstrumentationAnalysis(); });

  LoopAnalysis LIA;
  LoopInfo &&LI = LIA.run(*F, FAM);

  Loop *L = LI.getLoopFor(CanonicalLoop->getHeader());
```

Getting the LLVM Loop from the `CanonicalLoopInfo` struct

2. Attaching the metadata

```
// Add access group metadata to memory-access instructions.
MDNode *AccessGroup = MDNode::getDistinct(Ctx, {});
for (BasicBlock *BB : Reachable)
  addSimdMetadata(BB, AccessGroup, LI);
```

```
/// Attach llvm.access.group metadata to the memref instructions of \p Block
static void addSimdMetadata(BasicBlock *Block, MDNode *AccessGroup,
                            LoopInfo &LI) {
  for (Instruction &I : *Block) {
    if (I.mayReadOrWriteMemory()) {
      // TODO: This instruction may already have access group from
      // other pragmas e.g. #pragma clang loop vectorize.  Append
      // so that the existing metadata is not overwritten.
      I.setMetadata(LLVMContext::MD_access_group, AccessGroup);
    }
  }
}
```

HUAWEI

2. Attaching the metadata

```
!5 = distinct !{}
!6 = distinct !{!6, !7, !8}
!7 = !{!"llvm.loop.parallel_accesses", !5}
!8 = !{!"llvm.loop.vectorize.enable", i1 true}
```

```
// Use the above access group metadata to create loop level
// metadata, which should be distinct for each loop.
ConstantAsMetadata *BoolConst =
    ConstantAsMetadata::get(ConstantInt::getTrue(Type::getInt1Ty(Ctx)));
// TODO:  If the loop has existing parallel access metadata, have
// to combine two lists.
addLoopMetadata(
    CanonicalLoop,
    {MDNode::get(Ctx, {MDString::get(Ctx, "llvm.loop.parallel_accesses"),
                       AccessGroup}),
     MDNode::get(Ctx, {MDString::get(Ctx, "llvm.loop.vectorize.enable"),
                       BoolConst})});
```

Make sure that the access group metadata is unique to each SIMD loop

Add test cases

clang/test/OpenMP/irbuilder_simd.cpp

→ llvm-lit test to check if the expected IR is generated by clang

```
  int a, b;
};

void simple(float *a, float *b, int *c) {
  S s, *p;
  P pp;
#pragma omp simd
  for (int i = 3; i < 32; i += 5) {
    // llvm.access.group test
    // CHECK: %[[A_ADDR:.+]] = alloca float*, align 8
    // CHECK: %[[B_ADDR:.+]] = alloca float*, align 8
    // CHECK: %[[S:.+]] = alloca %struct.S, align 4
    // CHECK: %[[P:.+]] = alloca %struct.S*, align 8
    // CHECK: %[[I:.+]] = alloca i32, align 4
    // CHECK: %[[TMP3:.+]] = load float*, float** %[[B_ADDR:.+]], align 8, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[TMP4:.+]] = load i32, i32* %[[I:.+]], align 4, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[IDXPROM:.+]] = sext i32 %[[TMP4:.+]] to i64
    // CHECK-NEXT: %[[ARRAYIDX:.+]] = getelementptr inbounds float, float* %[[TMP3:.+]], i64 %[[IDXPROM:.+]]
    // CHECK-NEXT: %[[TMP5:.+]] = load float, float* %[[ARRAYIDX:.+]], align 4, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[A2:.+]] = getelementptr inbounds %struct.S, %struct.S* %[[S:.+]], i32 0, i32 0
    // CHECK-NEXT: %[[TMP6:.+]] = load i32, i32* %[[A2:.+]], align 4, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[CONV:.+]] = sitofp i32 %[[TMP6:.+]] to float
    // CHECK-NEXT: %[[ADD:.+]] = fadd float %[[TMP5:.+]], %[[CONV:.+]]
    // CHECK-NEXT: %[[TMP7:.+]] = load %struct.S*, %struct.S** %[[P:.+]], align 8, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[A3:.+]] = getelementptr inbounds %struct.S, %struct.S* %[[TMP7:.+]], i32 0, i32 0
    // CHECK-NEXT: %[[TMP8:.+]] = load i32, i32* %[[A3:.+]], align 4, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[CONV4:.+]] = sitofp i32 %[[TMP8:.+]] to float
    // CHECK-NEXT: %[[ADD5:.+]] = fadd float %[[ADD:.+]], %[[CONV4:.+]]
    // CHECK-NEXT: %[[TMP9:.+]] = load float*, float** %[[A_ADDR:.+]], align 8, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[TMP10:.+]] = load i32, i32* %[[I:.+]], align 4, !llvm.access.group ![[META3:[0-9]+]]
    // CHECK-NEXT: %[[IDXPROM6:.+]] = sext i32 %[[TMP10:.+]] to i64
    // CHECK-NEXT: %[[ARRAYIDX7:.+]] = getelementptr inbounds float, float* %[[TMP9:.+]], i64 %[[IDXPROM6:.+]]
    // CHECK-NEXT: store float %[[ADD5:.+]], float* %[[ARRAYIDX7:.+]], align 4, !llvm.access.group ![[META3:[0-9]+]]
    // llvm.loop test
    // CHECK: %[[OMP_LOOPDOTNEXT:.+]] = add nuw i32 %[[OMP_LOOPDOTIV:.+]], 1
    // CHECK-NEXT: br label %omp_loop.header, !llvm.loop ![[META4:[0-9]+]]
    a[i] = b[i] + s.a + p->a;
  }
```

HUAWEI

Add test cases

- `llvm/unittests/Frontend/OpenMPIRBuilderTest.cpp`
- → Calls your implemented functions then verifies modules etc.

```
TEST_F(OpenMPIRBuilderTest, ApplySimd) {
  OpenMPIRBuilder OMPBuilder(*M);

  CanonicalLoopInfo *CLI = buildSingleLoopFunction(DL, OMPBuilder);

  // Simd-ize the loop.
  OMPBuilder.applySimd(DL, CLI);

  OMPBuilder.finalize();
  EXPECT_FALSE(verifyModule(*M, &errs()));

  PassBuilder PB;
  FunctionAnalysisManager FAM;
  PB.registerFunctionAnalyses(FAM);
  LoopInfo &LI = FAM.getResult<LoopAnalysis>(*F);

  const std::vector<Loop *> &TopLvl = LI.getTopLevelLoops();
  EXPECT_EQ(TopLvl.size(), 1u);

  Loop *L = TopLvl.front();
  EXPECT_TRUE(findStringMetadataForLoop(L, "llvm.loop.parallel_accesses"));
  EXPECT_TRUE(getBooleanLoopAttribute(L, "llvm.loop.vectorize.enable"));

  // Check for llvm.access.group metadata attached to the printf
  // function in the loop body.
  BasicBlock *LoopBody = CLI->getBody();
  EXPECT_TRUE(any_of(*LoopBody, [](Instruction &I) {
    return I.getMetadata("llvm.access.group") != nullptr;
  }));
}

TEST_F(OpenMPIRBuilderTest, UnrollLoopFull) {
```

Steps for implementation

1. Read about the behavior of the construct/clause from OpenMP website
2. Identify the IR changes necessary
3. Identify if IRBuilder support is needed, implement
4. Define/modify OpenMP MLIR Op

HUAWEI

MLIR Operation definition

MLIR Operation definition

- Declaratively define OpenMP operations
  - Uses tablegen
- Can define input and output operands
- Whether operations have regions inside them
- Generic or custom printers and parser
- In the file `mlir/include/mlir/Dialect/OpenMP/OpenMPOps.td`

HUAWEI

SIMD Operation definition

```
def SimdLoopOp : OpenMP_Op<"simdloop", [AttrSizedOperandSegments,
                         AllTypesMatch<["lowerBound", "upperBound", "step"]>]> {
 let summary = "simd loop construct";
  let description = [{
    The simd construct can be applied to a loop to indicate that the loop can be
    ...
    ```
    omp.simdloop (%i1, %i2) : index = (%c0, %c0) to (%c10, %c10)
                                  step (%c1, %c1) {
      omp.yield
    }
    ```
  }];

  // TODO: Add other clauses
  let arguments = (ins Variadic<IntLikeType>:$lowerBound,
             Variadic<IntLikeType>:$upperBound,
             Variadic<IntLikeType>:$step);

  let regions = (region AnyRegion:$region);

  let extraClassDeclaration = [{
    /// Returns the number of loops in the simd loop nest.
    unsigned getNumLoops() { return lowerBound().size(); }

  }];

  let hasCustomAssemblyFormat = 1;
  let hasVerifier = 1;
}
```

Parser, Custom printer and verifier

- **mlir/lib/Dialect/OpenMP/IR/OpenMPDialect.cpp**

```
omp.simdloop (%i1, %i2) : i32 = (%c0, %c0)
  to (%c10, %c10) step (%c1, %c1) {
        ...
}
```

```cpp
ParseResult SimdLoopOp::parse(OpAsmParser &parser, OperationState &result) {
  // Parse an opening `(` followed by induction variables followed by `)`
  SmallVector<OpAsmParser::OperandType> ivs;
  if (parser.parseRegionArgumentList(ivs, /*requiredOperandCount=*/-1,
                                     OpAsmParser::Delimiter::Paren))
    return failure();
  int numIVs = static_cast<int>(ivs.size());
  Type loopVarType;
  if (parser.parseColonType(loopVarType))
    return failure();
  // Parse loop bounds.
  SmallVector<OpAsmParser::OperandType> lower;
  if (parser.parseEqual() ||
      parser.parseOperandList(lower, numIVs, OpAsmParser::Delimiter::Paren) ||
      parser.resolveOperands(lower, loopVarType, result.operands))
    return failure();
  SmallVector<OpAsmParser::OperandType> upper;
  if (parser.parseKeyword("to") ||
      parser.parseOperandList(upper, numIVs, OpAsmParser::Delimiter::Paren) ||
      parser.resolveOperands(upper, loopVarType, result.operands))
    return failure();

  // Parse step values.
  SmallVector<OpAsmParser::OperandType> steps;
  if (parser.parseKeyword("step") ||
      parser.parseOperandList(steps, numIVs, OpAsmParser::Delimiter::Paren) ||
      parser.resolveOperands(steps, loopVarType, result.operands))
    return failure();
```

Parser, Custom printer and verifier

```
omp.simdloop (%i1, %i2) : i32=
 (%c0, %c0) to (%c10, %c10)
             step (%c1, %c1) {
             ...
}
```

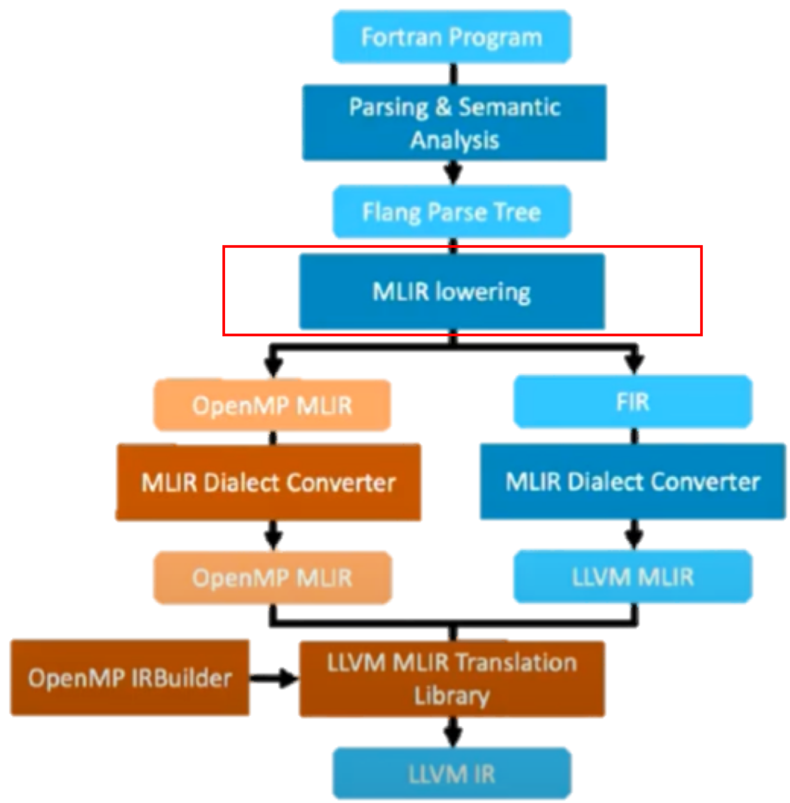
```cpp
void SimdLoopOp::print(OpAsmPrinter &p) {
  auto args = getRegion().front().getArguments();
  p << " (" << args << ") : " << args[0].getType() << " = (" << lowerBound()
    << ") to (" << upperBound() << ") ";
  p << "step (" << step() << ") ";

  p.printRegion(region(), /*printEntryBlockArgs=*/false);
}


//===----------------------------------------------------------------------==
// Verifier for Simd construct [2.9.3.1]
//===----------------------------------------------------------------------==

LogicalResult SimdLoopOp::verify() {
  if (this->lowerBound().empty()) {
    return emitOpError() << "empty lowerbound for simd loop operation";
  }
  return success();
}
```

Steps for implementation

1. Read about the behavior of the construct/clause from OpenMP website
2. Identify the IR changes necessary
3. Identify if IRBuilder support is needed, implement
4. Define/modify OpenMP MLIR Op
5. Verify definition by implementing lowering

HUAWEI

Verify definition by implementing lowering



Verifying your MLIR definition

- flang/lib/**lower**/OpenMP.cpp
- genOMP() function works on various Fortran::parser::<Construct> types

```
void Fortran::lower::genOpenMPConstruct(
    Fortran::lower::AbstractConverter &converter,
    Fortran::lower::pft::Evaluation &eval,
    const Fortran::parser::OpenMPConstruct &ompConstruct) {

  std::visit(
      common::visitors{
          [&](const Fortran::parser::OpenMPStandaloneConstruct
                  &standaloneConstruct) {
            genOMP(converter, eval, standaloneConstruct);
          },
          [&](const Fortran::parser::OpenMPSectionsConstruct
                  &sectionsConstruct) {
            TODO(converter.getCurrentLocation(), "OpenMPSectionsConstruct");
          },
          [&](const Fortran::parser::OpenMPLoopConstruct &loopConstruct) {
            genOMP(converter, eval, loopConstruct);
          },
          [&](const Fortran::parser::OpenMPDeclarativeAllocate
                  &execAllocConstruct) {
            TODO(converter.getCurrentLocation(), "OpenMPDeclarativeAllocate");
          },
          [&](const Fortran::parser::OpenMPExecutableAllocate
                  &execAllocConstruct) {
            TODO(converter.getCurrentLocation(), "OpenMPExecutableAllocate");
```

Check the parse tree for construct type

```fortran
program simdloop
        integer :: i
!$OMP SIMD
do i=1, 9
  print*, i
end do
!$OMP END SIMD
end
```

```
Program -> ProgramUnit -> MainProgram
| ProgramStmt -> Name = 'simdloop'
| SpecificationPart
| | ImplicitPart ->
| | DeclarationConstruct -> SpecificationConstruct -> TypeDeclarationStmt
| | | DeclarationTypeSpec -> IntrinsicTypeSpec -> IntegerTypeSpec ->
| | | EntityDecl
| | | | Name = 'i'
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct -> OpenMPConstruct -> OpenMPLoopConstruct
| | | OmpBeginLoopDirective
| | | | OmpLoopDirective -> llvm::omp::Directive = simd
| | | | OmpClauseList ->
| | | DoConstruct
| | | | NonLabelDoStmt
| | | | | LoopControl -> LoopBounds
| | | | | | Scalar -> Name = 'i'
| | | | | | Scalar -> Expr = '1_4'
| | | | | | | LiteralConstant -> IntLiteralConstant = '1'
| | | | | | Scalar -> Expr = '9_4'
| | | | | | | LiteralConstant -> IntLiteralConstant = '9'
| | | | Block
| | | | | ExecutionPartConstruct -> ExecutableConstruct -> ActionStmt -> PrintStmt
```

```
./f18-llvm-project/build/bin/flang-new -fc1 -fopenmp -fsyntax-only -fdebug-dump-parse-tree ./omp-loop.f90 -o -
```

HUAWEI

Creating the SimdLoop operation in genOMP()

- Extract `lowerbound`, `upperbound`, `step` (optional) from the parse tree
- Use the extracted info to create a new `SimdLoopOp`
- Generate the body (region) that b.elongs inside the `SimdLoopOp`

Creating the SimdLoop operation in genOMP()

```
omp.simdloop (%i1) : i32=  (%c1) to (%c19) step (%c1) {
        <region>
}
```

```
auto *doStmt = doLoop->getIf<Fortran::parser::NonLabelDoStmt>();
assert(doStmt && "Expected do loop to be in the nested evaluation");
const auto &loopControl =
    std::get<std::optional<Fortran::parser::LoopControl>>(doStmt->t);
const Fortran::parser::LoopControl::Bounds *bounds =
    std::get_if<Fortran::parser::LoopControl::Bounds>(&loopControl->u);
if (bounds) {
  Fortran::lower::StatementContext stmtCtx;
  lowerBound.push_back(fir::getBase(converter.genExprValue(
      *Fortran::semantics::GetExpr(bounds->lower), stmtCtx)));
  upperBound.push_back(fir::getBase(converter.genExprValue(
      *Fortran::semantics::GetExpr(bounds->upper), stmtCtx)));
  if (bounds->step) {
    step.push_back(fir::getBase(converter.genExprValue(
        *Fortran::semantics::GetExpr(bounds->step), stmtCtx)));
  }
```

```
auto SimdLoopOp = firOpBuilder.create<mlir::omp::SimdLoopOp>(
  currentLocation, resultType, lowerBound, upperBound, step);
createBodyOfOp<omp::SimdLoopOp>(SimdLoopOp, converter, currentLocation, eval,
                                &wsLoopOpClauseList, iv);
```

```
Program -> ProgramUnit -> MainProgram
| ProgramStmt -> Name = 'simdloop'
| SpecificationPart
| | ImplicitPart ->
| | DeclarationConstruct -> SpecificationConstruct -> TypeD
| | | DeclarationTypeSpec -> IntrinsicTypeSpec -> IntegerTy
| | | EntityDecl
| | | | Name = 'i'
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct -> OpenMP
| | | OmpBeginLoopDirective
| | | | OmpLoopDirective -> llvm::omp::Directive = simd
| | | | OmpClauseList ->
| | | DoConstruct
| | | | NonLabelDoStmt
| | | | | LoopControl -> LoopBounds
| | | | | | Scalar -> Name = 'i'
| | | | | | Scalar -> Expr = '1_4'
| | | | | | | LiteralConstant -> IntLiteralConstant = '1'
| | | | | | Scalar -> Expr = '9_4'
| | | | | | | LiteralConstant -> IntLiteralConstant = '9'
| | | | | Block
| | | | | | ExecutionPartConstruct -> ExecutableConstruct ->
```

MLIR Verification final step

```fortran
program simdloop
        integer :: i
!$OMP SIMD
do i=1, 9
  print*, i
end do
!$OMP END SIMD
end
```

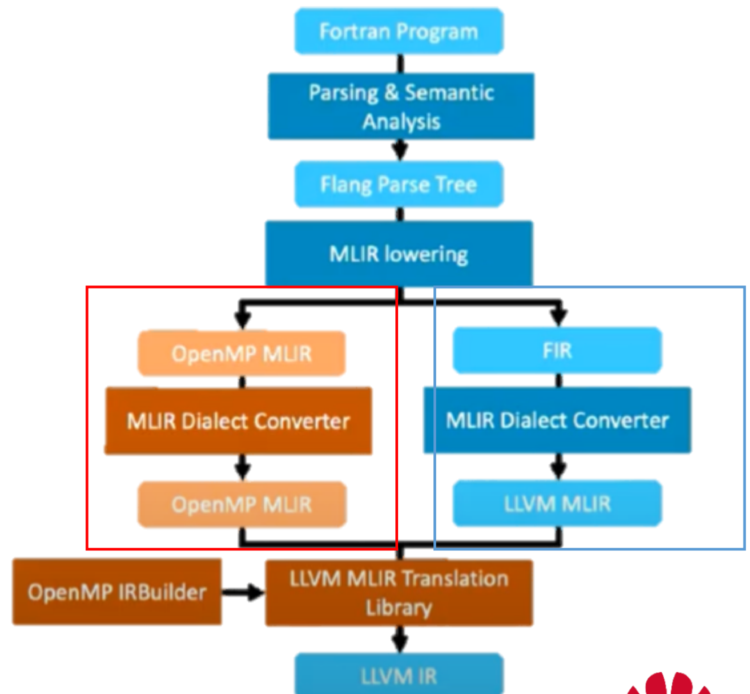
```
build/bin/bbc -fopenmp -emit-fir ./omp-loop.f90 -o -
```

```
func @_QQmain() {
  %0 = fir.alloca i32 {bindc_name = "i", uniq_name = "_QFEi"}
  %c1_i32 = arith.constant 1 : i32
  %c9_i32 = arith.constant 9 : i32
  %c1_i32_0 = arith.constant 1 : i32
  omp.simdloop (%arg0) : i32 = (%c1_i32) to (%c9_i32) step (%c1_i32_0) {
    %c-1_i32 = arith.constant -1 : i32
    %1 = fir.address_of(@_QQcl.2E2F2E2F6F6D702D6C6F6F702E66393000) : !fir.ref<!
    %2 = fir.convert %1 : (!fir.ref<!fir.char<1,17>>) -> !fir.ref<i8>
    %c5_i32 = arith.constant 5 : i32
    %3 = fir.call @_FortranAioBeginExternalListOutput(%c-1_i32, %2, %c5_i32) :
    %4 = fir.call @_FortranAioOutputInteger32(%3, %arg0) : (!fir.ref<i8>, i32)
    %5 = fir.call @_FortranAioEndIoStatement(%3) : (!fir.ref<i8>) -> i32
    omp.yield
  }
  return
}
```

HUAWEI

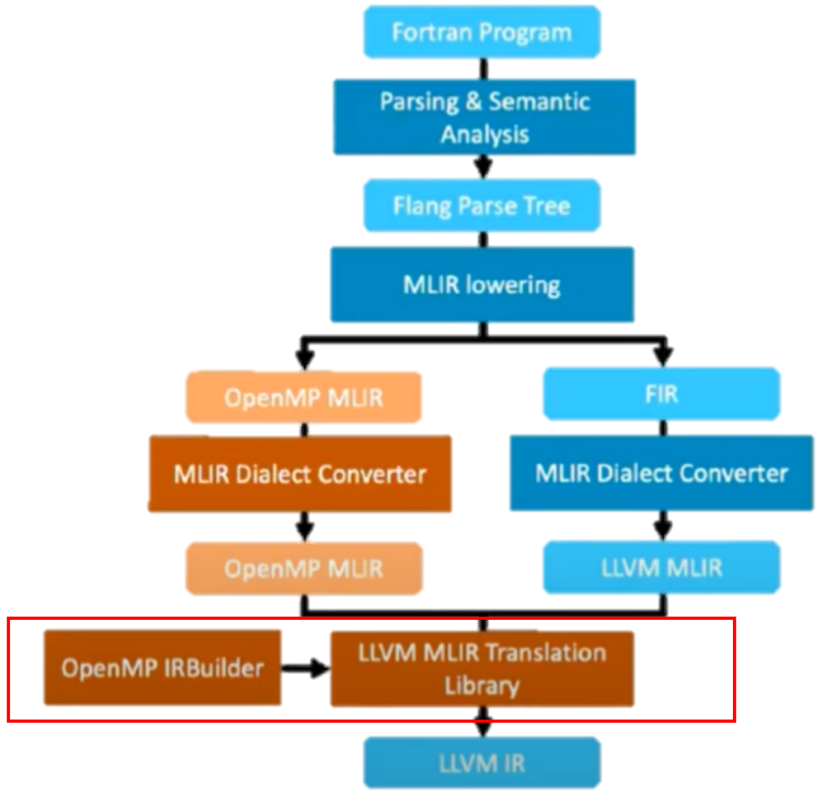
OpenMP MLIR <-> FIR co-existence

```
func @_QQmain() {
  %0 = fir.alloca i32 {bindc_name = "i", uniq_name = "_QFEi"}
  %c1_i32 = arith.constant 1 : i32
  %c9_i32 = arith.constant 9 : i32
  %c1_i32_0 = arith.constant 1 : i32
  omp.simdloop (%arg0) : i32 = (%c1_i32) to (%c9_i32) step (%c1_i32_0) {
    %c-1_i32 = arith.constant -1 : i32
    %1 = fir.address_of(@_QQcl.2E2F2E2F6F6D702D6C6F6F702E66393000) : !fir.ref<!
    %2 = fir.convert %1 : (!fir.ref<!fir.char<1,17>>) -> !fir.ref<i8>
    %c5_i32 = arith.constant 5 : i32
    %3 = fir.call @_FortranAioBeginExternalListOutput(%c-1_i32, %2, %c5_i32) :
    %4 = fir.call @_FortranAioOutputInteger32(%3, %arg0) : (!fir.ref<i8>, i32)
    %5 = fir.call @_FortranAioEndIoStatement(%3) : (!fir.ref<i8>) -> i32
    omp.yield
  }
  return
}
```



HUAWEI

Lowering the MLIR to LLVMIR



Lowering the MLIR to LLVMIR

- **`OpenMPToLLVMIRTranslation.cpp`**

```
/// Given an OpenMP MLIR operation, create the corresponding LLVM IR
/// (including OpenMP runtime calls).
LogicalResult OpenMPDialectLLVMIRTranslationInterface::convertOperation(
    Operation *op, llvm::IRBuilderBase &builder,
    LLVM::ModuleTranslation &moduleTranslation) const {

  llvm::OpenMPIRBuilder *ompBuilder = moduleTranslation.getOpenMPBuilder();

  return llvm::TypeSwitch<Operation *, LogicalResult>(op)
      .Case([&](omp::OrderedOp) {
        return convertOmpOrdered(*op, builder, moduleTranslation);
      })
      .Case([&](omp::WsLoopOp) {
        return convertOmpWsLoop(*op, builder, moduleTranslation);
      })
      .Case([&](omp::SimdLoopOp) {
        return convertOmpSimdLoop(*op, builder, moduleTranslation);
      })
      .Case([&](omp::AtomicReadOp) {
        return convertOmpAtomicRead(*op, builder, moduleTranslation);
      })
```

HUAWEI

Lowering the MLIR to LLVMIR

- Extract lower, upper bound and step from the MLIR `SimdLoopOp`
- Use the extracted values to generate LLVM IR using the `createCanonicalLoop`() API
- Add metadata using the `applySimd`() API from OMPIRBuilder.

HUAWEI

MLIR -> LLVM IR

```
llvm::OpenMPIRBuilder *ompBuilder = moduleTranslation.getOpenMPBuilder();
for (unsigned i = 0, e = loop.getNumLoops(); i < e; ++i) {
  llvm::Value *lowerBound =
      moduleTranslation.lookupValue(loop.lowerBound()[i]);
  llvm::Value *upperBound =
      moduleTranslation.lookupValue(loop.upperBound()[i]);
  llvm::Value *step = moduleTranslation.lookupValue(loop.step()[i]);

  }
  loopInfos.push_back(ompBuilder->createCanonicalLoop(
      loc, bodyGen, lowerBound, upperBound, step,
      /*IsSigned=*/true, /*Inclusive=*/true, computeIP));

  if (failed(bodyGenStatus))
    return failure();
}

ompBuilder->applySimd(loopInfo);

builder.restoreIP(afterIP);
return success();
```

Lowering to LLVMIR: TestCases

- Check for invalid operations (e.g. check if lb, ub and step has same type of not) → `mlir/test/Dialect/OpenMP/invalid.mlir`

```
func @omp_simdloop(%lb : index, %ub : index, %step : i32) -> () {
  // expected-error @below {{op failed to verify that all of {lowerBound, upp
  "omp.simdloop" (%lb, %ub, %step) ({
    ^bb0(%iv: index):
      omp.yield
  }) {operand_segment_sizes = dense<[1,1,1]> : vector<3xi32>} :
    (index, index, i32) -> ()

  return
}
```

- Check if printing etc is looking good → `mlir/test/Dialect/OpenMP/ops.mlir`

Lowering to LLVMIR: TestCases

- Check if the MLIR->LLVM IR translation is looking good
  - `mlir/test/Target/LLVMIR/openmp-llvm.mlir` **(uses mlir-translate)**

```
// CHECK-LABEL: @simdloop_simple
llvm.func @simdloop_simple(%lb : i64, %ub : i64, %step : i64, %arg0: !llvm.ptr<f32>) {
  "omp.simdloop" (%lb, %ub, %step) ({
    ^bb0(%iv: i64):
      %3 = llvm.mlir.constant(2.000000e+00 : f32) : f32
      // CHECK: llvm.access.group
      %4 = llvm.getelementptr %arg0[%iv] : (!llvm.ptr<f32>, i64) -> !llvm.ptr<f32>
      llvm.store %3, %4 : !llvm.ptr<f32>
      omp.yield
  }) {operand_segment_sizes = dense<[1,1,1]> : vector<3xi32>} :
    (i64, i64, i64) -> ()

  llvm.return
}
// CHECK: llvm.loop.parallel_accesses
// CHECK-NEXT: llvm.loop.vectorize.enable
```

Summary of lowering process

- Study up the operation
- Write a simple test case and look at the generated IR
- Check if OMPIRBuilder support is necessary (both clang and flang uses it) (patch 1)
- Define/modify OpenMP MLIR Op definitions, implement lowering (patch 2)
- Write proper test cases for both patches

HUAWEI

Thank you, Questions?

- Getting in touch
  - Technical calls
  - flang-dev mailing list
  - Join our slack channel flang-compiler.slack.com
  - Check this webpage for links (https://prereleases.llvm.org/11.0.0/rc3/tools/flang/docs/GettingInvolved.html)