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/1FvHPuSkGBi4mQZRAwChdvGx0dGboffI6-x06dqyqU0/edit#gid=0
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
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 reusable
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`
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]
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

```c
void omp_simd() {
  int i = 0;
  int a[16];
  #pragma omp simd
  for (int i=0; i <16; i++) {
    a[i] = i;
  }
  return;
}
```
Visualize the changes necessary in the final IR.
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
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 \(\rightarrow\) yes`

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

`[!$omp end do simd`  `Clause \(\rightarrow\) No`
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.

* for Clang
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
1. Creating the canonical loop

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

```cpp
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

Function that is called while lowering the SIMD directive

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

Lambda call

Lowering code

Uses `createCanonicalLoop()`

CGM \( \rightarrow \) per module state
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.
2. Attaching the metadata

Getting the LLVM Loop from the CanonicalLoopInfo struct

```c
void OpenMPTRBuilder::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());
}
```
2. Attaching the metadata

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

/// 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);
        }
    }
}
```
2. Attaching the metadata

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
Add test cases

- `llvm/unittests/Frontend/OpenMPIRBuilderTest.cpp`
- → Calls your implemented functions then verifies modules etc.
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
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
SIMD Operation definition
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) {
  ...
}
```
omp.simdloop (%i1, %i2) : i32= (%c0, %c0) to (%c10, %c10) step (%c1, %c1) {
  ...
}

Parser, Custom printer and verifier

```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
Verify definition by implementing lowering
Verifying your MLIR definition

- flang/lib/\texttt{lower}/OpenMP.cpp
- genOMP() function works on various Fortran::parser::<Construct> types
Check the parse tree for construct type

```
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 ./MainProgram
```
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()

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

```cpp
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))));
    }
    SimdLoopOp = firOpBuilder.create<mlir::omp::SimdLoopOp>(
        currentLocation, resultType, lowerBound, upperBound, step);
    createBodyOfOp<
```
MLIR Verification final step
OpenMP MLIR <-> FIR co-existence
Lowering the MLIR to LLVMIR
Lowering the MLIR to LLVMIR

• OpenMPToLLVMIRTranslation.cpp

```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::OpenMPBuilder *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); })
  };
```
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.
MLIR -> LLVM IR

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

• 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)*
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
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)