OpenMP in Flang using MLIR

LLVM Compiler and Tools for HPC
ISC-HPC 2020

Kiran Chandramohan
25 June 2020
Contents

• Introduction
• Flang compiler flow
• OpenMP support in Flang
• OpenMP plan for Flang
  • OpenMP Parse Tree representation
  • OpenMP Semantic Checks
  • OpenMP Operation Definition
  • Lowering to OpenMP dialect
  • Lowering to LLVM IR
• Status
• How to get involved
Introduction

- The Flang Fortran frontend was merged into LLVM on April 9
  - Flang started off as the F18 project at Nvidia in collaboration with US DoE
  - Arm, AMD and US DoE labs and a few individuals are contributing
  - Intends to replace the old Flang project (github.com/flang-compiler/flang)

- Built using modern technologies
  - Written in C++17
  - Uses MLIR

- Flang is a work in progress
  - Currently Flang performs parsing and semantic checks when invoked
  - It then unparses to Fortran
  - Searches for an external compiler to complete the compilation
    - Note: This is for testing
Flang compiler flow

- Parses Fortran 2018
- Performs semantic checks
- Lowers to a high level IR, FIR
  - Uses the MLIR framework
  - Come to this later
- Converts to a lower level IR, LLVM MLIR
- Lowers to LLVM IR

Diagram:

1. Fortran Program
2. Parsing & Semantic Analysis
3. Flang Parse Tree
4. MLIR lowering
5. FIR
6. MLIR Dialect Converter
7. LLVM MLIR
8. LLVM MLIR Translation Library
9. LLVM IR
OpenMP support in Flang

• Support for latest OpenMP standards is important in HPC
  • Latest published standard is OpenMP 5.0
  • OpenMP 5.1 to be announced later this year

• Support for latest OpenMP standards is important for Flang to enter production
  • Old Flang (flang-project/flang) has partial support for OpenMP 4.5

• What is supported in Flang now?
  • OpenMP 4.5 parsing
  • Semantic Checks (in progress)
  • Use --fopenmp flag to enable OpenMP

• Uses two components for OpenMP codegen
  • MLIR
  • OpenMP IRBuilder
MLIR

- Multi Level Intermediate Representation
- A new approach for building compiler infrastructure
  - Can use to build SSA-based Intermediate Representations (IRs)
  - Provides a declarative system for defining IRs
  - Provides common infrastructure (printing, parsing, location tracking, pass management etc)
- Flang compiler uses the 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 constructs
  - Makes OpenMP codegen re-usable
MLIR

- Operations in the IR can contain regions
- LLVM IR instructions cannot
- Representation in LLVM IR involves outlining

```
//MLIR
omp.parallel {
    %3 = llvm.add %1, %2 : !llvm.float
    omp.terminator
}

//LLVM IR
define @outlined_parallel(...) {
    ...
    %1 = fadd float %2, %3
    ...
}
call kmvc_fork_call(..., outlined_parallel,...)
```
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 the LLVM directory
  • llvm/lib/Frontend/OpenMP
Example: OpenMP Parallel

Fortran source with OpenMP

!Fortran code
!$omp parallel
  c = a + b
!$omp end parallel
!More Fortran code

Flang parse tree

<Fortran parse tree>
|     | ExecutionPartConstruct ->
|     | ExecutableConstruct ->
|     | OpenMPCconstruct ->
|     | OpenMPBlockConstruct
|     |     | OmpBlockDirective -> Directive = Parallel
|     |     | OmpClauseList ->
|     |     | Block
|     |     | ExecutionPartConstruct ->
|     |     | ExecutableConstruct -> ActionStmt ->
|     |     | AssignmentStmt
|     |     |     | Variable -> Designator ->
|     |     |     | DataRef -> Name = 'c'
|     |     |     |     | Expr -> Add
|     |     |     |     |     | Expr -> Designator -> DataRef
|     |     |     |     |     |     | Name = 'a'
|     |     |     |     |     | Expr -> Designator -> DataRef
|     |     |     |     |     |     | Name = 'b'
|     |     | OmpEndBlockDirective ->
|     |     | OmpBlockDirective -> Directive = Parallel <More Fortran parse tree>

MLIR: FIR + OpenMP

Mlir.region(...) {
  ...
  omp.parallel {
    %1 = fir.addf %2, %3 : fir.real<32>
  }
  %21 = <more fir> ...
}
Example: OpenMP Parallel

**MLIR: LLVM + OpenMP dialect**

Mlir.region(...)  
{  
...  
omp.parallel {  
  %1 = llvm.fadd %2, %3 : !llvm.float  
}  
%21 = <more llvm dialect>  
...  
}

**LLVM IR**

```mlir```
define @outlined_parallel_fn(...)  
{  
...  
%1 = fadd float %2, %3  
...  
}  
define @xyz(...)  
{  
%1 = alloca float  
....  
call  
kmpc_fork_call(...,outlined_parallel_fn,...)  
}
```
OpenMP plan for Flang

- Fortran Program
- Parsing & Semantic Analysis
- Flang Parse Tree
- MLIR lowering
  - OpenMP MLIR
    - MLIR Dialect Converter
    - OpenMP IRBuilder
  - LLVM MLIR Translation Library
    - LLVM IR

- FIR
- MLIR Dialect Converter
- LLVM MLIR
OpenMP Parse Tree representation

- OpenMP constructs are represented in the parse tree as
  - Executable Constructs: OpenMPConstruct
  - Declarative Constructs: OpenMPDeclarativeConstruct
- Flang uses variants in the parse tree representation

```c
struct OpenMPConstruct {
    UNION_CLASS_BOILERPLATE(OpenMPConstruct);
    std::variant<
        OpenMPStandaloneConstruct, OpenMPSectionsConstruct,
        OpenMPLoopConstruct, OpenMPBlockConstruct, OpenMPAtomicConstruct,
        OpenMPCriticalConstruct>
        u;
};
```
Flang parse tree with OpenMP

Fortran source

```
program mn
...
!$omp flush(arr)
...
end
```

Flang Parse tree

```
Program -> ProgramUnit -> MainProgram
| ProgramStmt -> Name = 'mn'
| SpecificationPart
| | ...
| ExecutionPart -> Block
| | ExecutionPartConstruct -> ExecutableConstruct ->
| | OpenMPConstruct -> OpenMPStandaloneConstruct ->
| | OpenMPFlushConstruct
| | | Verbatim
| | | OmpObjectList -> OmpObject -> Designator ->
| | | DataRef -> Name = 'arr'
| | ...
| EndProgramStmt ->
```
Flang parse tree with OpenMP: Tooling

• Visitor Class
class OpenMPCounter {
    template<typename A> bool Pre(const A &) { return true; }
    template<typename A> void Post(const A &) {}
    void Post(const Fortran::parser::OpenMPConstruct &) {counter++;}
    int counter{0};
}

• Usage
OpenMPCounter visitor;
void OpenMPStatisticsParseTree(const Fortran::parser::Program &program) {
    Fortran::parser::Walk(program, visitor);
}
OpenMP Semantic Checks

• Checks to ensure that Constructs and Clauses conform to the standard.
  • Permitted clauses in a construct
  • Clauses not occurring together
  • Specifying that expressions evaluate to a positive integer
  • Nesting checks

```cpp
void OmpStructureChecker::Enter(const parser::OpenMPDeclareSimdConstruct &x) {
  const auto &dir{std::get<parser::Verbatim>(x.t)};
  PushContext(dir.source, OmpDirective::DECLARE_SIMD);
  OmpClauseSet allowed{
      OmpClause::LINEAR, OmpClause::ALIGNED, OmpClause::UNIFORM};
  SetContextAllowed(allowed);
  SetContextAllowedOnce({OmpClause::SIMDLEN});
  SetContextAllowedExclusive({OmpClause::INBRANCH, OmpClause::NOTINBRANCH});
} imshow
```
OpenMP Operation Definition
MLIR: Operation Definition

• Declaratively define OpenMP operations
  • Uses tablegen
• Can define the input and output operands
• Whether operations have regions inside them
• Provides generic printers and parsers for operations
• Simple example of barrier operation in the next slide
OpenMP barrier construct : Definition

def OpenMP_Dialect : Dialect {
    let name = "omp";
}

class OpenMP_Op<string mnemonic, list<OpTrait> traits = []> : Op<OpenMP_Dialect, mnemonic, traits>;

def BarrierOp : OpenMP_Op"barrier"> {
    let summary = "barrier construct";
    let description = [{
        The barrier construct specifies an explicit barrier at the point at which the construct appears.
    }];

    let assemblyFormat = “attr-dict”;
}
MLIR: Customized Op Definition

• Sometimes custom printers and parsers are required
• This helps to define operations in a domain specific way
• OpenMP clauses are best defined as in a directive
• Clauses can have a variable number of arguments
• Definition of parallel operation in the next slide
  • Clauses are modeled as arguments
  • Arguments are operands or attributes (constants)
  • Most OpenMP clauses are optional
  • OpenMP clauses can have a variable number of elements (like variables)
OpenMP Parallel Construct : Definition

    let summary = "parallel construct";
    let description = {{ The parallel construct includes a region of code which is to be executed by a team of threads.}};
    let arguments = (ins Optional<AnyType>:$if_expr_var,
                   Optional<AnyType>:$num_threads_var,
                   OptionalAttr<ClauseDefault>::$default_val,
                   Variadic<AnyType>:$private_vars,
                   Variadic<AnyType>:$firstprivate_vars,
                   Variadic<AnyType>:$shared_vars,
                   Variadic<AnyType>:$copyin_vars,
                   OptionalAttr<ClauseProcBind>::$proc_bind_val);

    let regions = (region AnyRegion:$region);

    let parser = {{ return parseParallelOp(parser, result); }};
    let printer = {{ return printParallelOp(p, *this); }};
}
OpenMP Parallel : Example

Standard types

```c
omp.parallel shared(%data_var : memref<i32>)
  copyin(%data_var : memref<i32>,
    %data_var : memref<i32>) {
    omp.parallel if(%if_cond : i32) {
      omp.terminator
    } omp.terminator
} omp.terminator
```

LLVM dialect types

```c
omp.parallel
  num_threads(%num_threads : !llvm.i32)
  proc_bind(master) {
    omp.terminator
  }
```
Lowering to OpenMP dialect

1. Fortran Program
2. Parsing & Semantic Analysis
3. Flang Parse Tree
4. MLIR lowering
5. OpenMP MLIR
6. MLIR Dialect Converter
7. OpenMP MLIR
8. LLVM MLIR Translation Library
9. LLVM IR
Lowering to OpenMP dialect

- Happens along with FIR lowering
- Lowering code in flang/lib/Lower/Bridge.cpp
  - Calls code in flang/lib/Lower/OpenMP.cpp

```cpp
void Fortran::lower::genOpenMPConstruct(
    Fortran::lower::AbstractConverter &,
    Fortran::lower::pft::Evaluation &,
    const Fortran::parser::OpenMPConstruct &)
```
Lowering to LLVM IR

1. Fortran Program
2. Parsing & Semantic Analysis
3. Flang Parse Tree
4. MLIR lowering
   - OpenMP MLIR
   - MLIR Dialect Converter
5. OpenMP IRBuilder
6. LLVM MLIR Translation Library
7. LLVM IR
Lowering to LLVM IR

- LLVM dialect in MLIR contains a list of functions
- Each function has a list of blocks
- Blocks have a list of operations
- OpenMP operations can have blocks inside
Lowering to LLVM IR

```cpp
LogicalResult
ModuleTranslation::convertOmpOperation(Operation &opInst,
                                       llvm::IRBuilder<> &builder) {
    if (!ompBuilder) {
        ompBuilder = std::make_unique<llvm::OpenMPIRBuilder>(*llvmModule);
        ompBuilder->initialize();
    }
    return llvm::TypeSwitch<Operation *, LogicalResult>(&opInst)
        .Case([&](omp::BarrierOp) {
            ompBuilder->CreateBarrier(builder.saveIP(), llvm::omp::OMPD_barrier);
            return success();
        })
        .Case([&](omp::TaskwaitOp) {
            ompBuilder->CreateTaskwait(builder.saveIP());
            return success();
        })
    ...
```
OpenMP barrier : Lowering
mlir-translate -mlir-to-llvmir test/Target/openmp-llvm.mlir

```mlir
define void @empty() !dbg !3 {
  %omp_global_thread_num = call i32 @__km MPC_global_thread_num(%struct.ident_t* @2)
call void @__kmPC_barrier(%struct.ident_t* @1, i32 %omp_global_thread_num)
ret void, !dbg !7
}

; Function Attrs: nounwind
declare i32 @__kmPC_global_thread_num(%struct.ident_t*) #0

; Function Attrs: inaccessiblemem_or_argmemonly
declare void @__kmPC_barrier(%struct.ident_t*, i32) #1

attributes #0 = { nounwind }
attributes #1 = { inaccessiblemem_or_argmemonly }

!llvm.dbg.cu = !{!0}
!llvm.module.flags = !{!2}
```
## Status

- Implementing vertically construct by construct
- Joint work with Nvidia, AMD, ANL, ORNL, LANL, BSC, Arm

| Parsing                  | OpenMP 4.5 complete  
|                         | OpenMP 5.0 in progress  
|                         | (Flush, Taskwait, Depends)  
| Semantic Checks         | Allowed clauses, Exclusive clauses,  
|                         | Integer properties  
|                         | Allowed nesting checks  
| Lowering to OpenMP Dialect | Waiting on Bridge code to arrive  
| OpenMP Dialect and LLVM IR lowering | Barrier, Flush, Taskwait, Taskyield  
|                         | complete  
|                         | Parallel, Master in progress  
| OpenMP IRBuilder        | Parallel and several constructs complete  
|                         | Sections, Target, Privatisation in progress  

© 2020 Arm Limited (or its affiliates)
How to get involved

• Project Management via google docs spreadsheet
• Separate sheets for parsing, semantics, OpenMP MLIR, lowerings, OpenMP IRBuilder
  • Currently has entries as per OpenMP 5.0
  • https://docs.google.com/spreadsheets/d/1FvHPuSkGbI4mQZRAwCIndvQx9dQboffiD-xD0qxoU0/edit#gid=0
• Weekly meeting on Thursday (4pm UK time)
  • https://docs.google.com/document/d/1yA-MeJf6RYY-ZXpdol0t7YoDoqtwAyBhFLr5thu5pFI/edit