Leveraging “nsw” in Flang’s LLVM IR Generation for Vectorization

Yusuke Minato

LLVM Developers’ Meeting – April 2025

EURO LLVM

Developers' Meeting

BERLIN 2025

Introduction

- Vectorization for Flang
  - Flang: the Fortran frontend in the LLVM project
  - Flang relies on the LoopVectorize pass in the backend.
    - Its frontend itself doesn't have a vectorization pass.
  - We're trying to improve the capability of vectorization in the backend.
    - Vectorization plays an important role in optimizations.
- TSVC (Test Suite for Vectorizing Compilers)
  - Available in both Fortran and C
  - Helps distinguish frontend problems from backend issues
    - The frontend can affect vectorization because it may generate LLVM IR that is difficult for the backend to vectorize.

Flang's Capability of Vectorization

FUJITSU

- Evaluation using TSVC
  - Options: `-O3 -ffast-math -march=armv9-a`
- Our contribution
  - Three additional loops can be vectorized since LLVM 20.
    - By introducing several options related to integer overflow into Flang

Vectorizable or not with		Number in LLVM 19	Number in LLVM 20
Clang	Flang		
Vectorizable	Vectorizable	76	80 (+4)
Vectorizable	**Non-Vectorizable**	**11**	**9 (-3+1)**
Non-Vectorizable	Vectorizable	1	1 (0)
Non-Vectorizable	Non-Vectorizable	40	38 (-2)

Analysis of the Result

- 11 loops fall into four categories:
  A) **Suboptimal analysis for array subscripts (3 loops)**
  B) Reduction variables passed by reference as real arguments (4 loops)
  C) Gather/scatter with non-constant strides at compile-time (3 loops)
  D) Loops requiring LTO only for Fortran version (1 loop)

- Categories A and B can be addressed in the frontend.
  - Adding more information to generated MLIR makes it easier for the LoopVectorize pass to vectorize the input LLVM IR.

Address Calculations in Fortran

FUJITSU

- Array subscripts: 32 bits, Addresses: 1 word
  - 1 word is equal to 64 bits on 64-bit CPUs.
- Address calculations frequently involve different bit lengths.
  - Lower bounds of subscripts are rarely 0, unlike in C.
    - Address calculations need a step to calculate offsets from subscripts.
- Internal representation of array subscripts gets complicated.
  - This can significantly influence analyses and loop vectorization.

```fortran
real a(n)
do i=1,n-1
  ... a(i+1) ...
end do
```

Address calculations are linearized because the shape of an array is often mutable in Fortran.

```llvm
%16 = load i32, ptr %3, align 4 ;; i
%17 = add i32 %16, 1 ;; i + 1
%18 = sext i32 %17 to i64
%19 = sub nsw i64 %18, 1 ;; subtract the lower bound
%20 = mul nsw i64 %19, 1 ;; multiply by the stride
%21 = mul nsw i64 %20, 1 ;; multiply by the size of lower dims
%22 = add nsw i64 %21, 0
%23 = mul nsw i64 1, %7
%24 = getelementptr float, ptr %0, i64 %22 ;; a + ((i + 1) - 1L)
%25 = load float, ptr %24, align 4 ;; a(i+1)
```

↑ subscript

↓ address

"nsw" in LLVM IR

- Attribute on `add/sub/mul/shl/trunc` in LLVM IR
  - Shows the result will not overflow the range of signed integer
- Use case of `nsw`
  - Simplifying calculations involving different bit lengths
    - `(i + 1) - 1L == i` ?
- Do not add `nsw` to instructions whose results could overflow.
  - cf. Rust (release mode)
    - The behavior of integer overflow is defined (wraparound).

“nsw” in LLVM IR

- Attribute on `add/sub/mul/shl/trunc` in LLVM IR
  - Shows the result will not overflow the range of signed integer
- Use case of `nsw`
  - Simplifying calculations involving different bit lengths
    - `(i + 1) - 1L == i` ?
      - Where `i` = `INT_MAX`, (LHS) = `(long)INT_MIN - 1L`, (RHS) = `(long)INT_MAX`
      - Equivalent only if the addition has an `nsw` flag (i.e., `i <= INT_MAX - 1`)

- Do not add `nsw` to instructions whose results could overflow.
  - cf. Rust (release mode)
    - The behavior of integer overflow is defined (wraparound).

Integer Overflow in Fortran

- The Fortran standard does not mention integer overflow.
  - Leaves the handling of integer overflow up to compiler developers

- We can assume that some operations will never overflow:
  - Increments of DO loop variables
  - Calculations for array subscripts
  - Calculations for loop bounds
- However, code that violates our assumption may exist.
  - The option `-fwrapv` has been introduced into Flang, similar to Clang.

Implementation Details

- Introduced a new flag in `FirOpBuilder`, a kind of IRBuilder
  - The Builder lowers the AST to IR recursively.
  - This flag controls whether `nsw` is added to the target operations.
    - Caution: Fortran 2008 and later have intrinsics for bitwise comparisons.
- Patches
  - #91579, #110060, #113854, #110061, #118933

```
   template <typename T>
   hlfir::Entity
   HlfirDesignatorBuilder::genSubscript(const Fortran::evaluate::Expr<T> &expr) {
+    fir::FirOpBuilder &builder = getBuilder();
+    mlir::arith::IntegerOverflowFlags iofBackup{};
+    if (!getConverter().getLoweringOptions().getIntegerWrapAround()) {
+      iofBackup = builder.getIntegerOverflowFlags();
+      builder.setIntegerOverflowFlags(mlir::arith::IntegerOverflowFlags::nsw);
+    }
     auto loweredExpr =
         HlfirBuilder(getLoc(), getConverter(), getSymMap(), getStmtCtx())
             .gen(expr);
-    fir::FirOpBuilder &builder = getBuilder();
+    if (!getConverter().getLoweringOptions().getIntegerWrapAround())
+      builder.setIntegerOverflowFlags(iofBackup);
```

```
+    auto iofi =
+        mlir::dyn_cast<mlir::arith::ArithIntegerOverflowFlagsInterface>(*op);
+    if (iofi) {
+      llvm::StringRef arithIOFAttrName = iofi.getIntegerOverflowAttrName();
+      if (integerOverflowFlags != mlir::arith::IntegerOverflowFlags::none)
+        op->setAttr(arithIOFAttrName,
+                    mlir::arith::IntegerOverflowFlagsAttr::get(
+                        op->getContext(), integerOverflowFlags));
+    }
```

LLVM IR Example with nsw Required

FUJITSU

```fortran
subroutine sample(a,lb,ub)
  integer lb,ub,i
  integer a(lb:ub)
  do i=lb,ub-1
    a(i+1) = i-lb
  end do
end subroutine
```

↓

```llvm
define void @sample_(ptr captures(none) %0, ptr captures(none) %1, ptr captures(none) %2) {
  %4 = load i32, ptr %1, align 4, !tbaa !4
  %5 = sext i32 %4 to i64
  %6 = load i32, ptr %2, align 4, !tbaa !10
  %7 = sext i32 %6 to i64
  %8 = sub i64 %7, %5
  %9 = add i64 %8, 1
  %10 = icmp sgt i64 %9, 0
  %11 = select i1 %10, i64 %9, i64 0
  %12 = sub nsw i32 %6, 1 ;; the upper bound of the loop
  %13 = sext i32 %12 to i64
  %14 = trunc i64 %5 to i32
  %15 = sub i64 %13, %5
  %16 = add i64 %15, 1
  br label %17

17:                                               ; preds = %21, %3
  %18 = phi i32 [ %32, %21 ], [ %14, %3 ]
  %19 = phi i64 [ %33, %21 ], [ %16, %3 ]
  %20 = icmp sgt i64 %19, 0
  br i1 %20, label %21, label %34

21:                                               ; preds = %17
  %22 = load i32, ptr %1, align 4, !tbaa !4
  %23 = sub i32 %18, %22
  %24 = add nsw i32 %18, 1 ;; the array subscript
  %25 = sext i32 %24 to i64
  %26 = sub nsw i64 %25, %5
  %27 = mul nsw i64 %26, 1
  %28 = mul nsw i64 %27, 1
  %29 = add nsw i64 %28, 0
  %30 = mul nsw i64 1, %11
  %31 = getelementptr i32, ptr %0, i64 %29
  store i32 %23, ptr %31, align 4, !tbaa !12
  %32 = add nsw i32 %18, 1 ;; the increment of the DO variable
  %33 = sub i64 %19, 1
  br label %17

34:                                               ; preds = %17
  ret void
}
```

Challenges

- Performance regression in the LoopStrengthReduce pass
  - While `sdiv` is changed to `udiv` in the InstCombine pass when considering `nsw` on its operands, it blocks analysis for the optimization.
    - https://github.com/llvm/llvm-project/issues/117318
- Remaining issues identified by TSVC (categories B and C)
  - Adding `nocapture` attribute to arguments
    - https://github.com/llvm/llvm-project/issues/106682
  - Accepting non-constant strides if they are found to be loop-invariant
    - https://github.com/llvm/llvm-project/issues/110611

Acknowledgements

- This presentation is based on results obtained from a project, JPNP21029, subsidized by the New Energy and Industrial Technology Development Organization (NEDO).

Thank you