Flang - the Fortran Compiler

Contents

- Flang – the Fortran Compiler
  - SYNOPSIS
  - DESCRIPTION
  - OPTIONS

SYNOPSIS

flang [*options*] *filename ...*

DESCRIPTION

Flang is the Fortran front-end designed for integration with LLVM and suitable for interoperability with Clang/LLVM. Flang consists of two components flang1 and flang2. Flang1 will be invoked by front end driver which is responsible for transforming the Fortran programs into tokens, then the parser transforms these tokens into Abstract Syntax Tree (AST). This AST is then transformed into canonical form which is used to generate ILM code. Then flang2 takes up this ILM code and transforms into ILI, which is then optimized by internal optimizer. The optimized ILI is then transformed in to LLVM IR. Then, the frontend driver transfers this LLVM IR to LLVM optimizer for optimization and target code generation.

Note: AOCC 2.2 extends the GitHub version found here with enhancements and stability

IEEE-754 Support

The Flang compiler does not conform to IEEE-754 specifications when -Ofast or -ffast-math options are specified. The compiler will enable a range of optimizations that provide faster mathematical operations under -Ofast and -ffast-math mode of compilation.

Code Generation and Optimization

Flang relies on AOCC's optimizer and code generator to transform the available LLVM IR and generate the best code for the target x86 platform.

OPTIONS

Compiler Options

For a list of compiler option, enter

- $flang -help
- $flang -help-hidden

The Flang compiler supports all clang compiler options as well as the following flang-specific compiler options

-Kieee – enabled by default from AOCC 2.2.0

Instructs the compiler to conform to the IEEE-754 specifications. The compiler will Perform floating-point operations in strict conformance with the IEEE 754 standard. Some optimizations are disabled when this option is specified

-Menable-vectorize-pragmas=<value>

Honours the vectorization pragmas specified in the fortran programs. The vectorization pragms vector, novector and ivdep are supported in this release.

-no-flang-libs

Do not link against Flang libraries

-mp

Enable OpenMP and link with with OpenMP library libomp

-nomp

Do not link with OpenMP library libomp

-Mbackslash

Treat backslash character like a C-style escape character

-Mno-backslash

Treat backslash like any other character

-Mbyteswapio

Swap byte-order for unformatted input/output

-Mfixed

Assume fixed-format source

-Mextend

Allow source lines up to 132 characters

-Mfreeform

Assume free-format source

-Mpreprocess

Run preprocessor for Fortran files

-Mstandard

Check standard conformance

-Msave

Assume all variables have SAVE attribute

-module

path to module file (-I also works)

-Mallocatable=95

Select Fortran 95 semantics for assignments to allocatable objects (Default)

-Mallocatable=03

Select Fortran 03 semantics for assignments to allocatable objects

-static-flang-libs

Link using static Flang libraries

-M[no]daz

Treat denormalized numbers as zero

-M[no]flushz

Set SSE to flush-to-zero mode

-Mcache_align

Align large objects on cache-line boundaries

-M[no]fprelaxed

This option is ignored

-fdefault-integer-8

Treat INTEGER and LOGICAL as INTEGER*8 and LOGICAL*8

-fdefault-real-8

Treat REAL as REAL*8

-i8

Treat INTEGER and LOGICAL as INTEGER*8 and LOGICAL*8

-r8

Treat REAL as REAL*8

-fno-fortran-main

Don't link in Fortran main

-Mrecursive

Allocate local variables on the stack, thus allowing recursion. SAVEd, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch

Target Selection Options

-march=<cpu>

Specify that flang should generate code for a specific processor family member and later. For example, if you specify -march=i486, the compiler is allowed to generate instructions that are valid on i486 and later processors, but which may not exist on earlier ones.

-march=znver1

Use this architecture flag for enabling best code generation and tuning for AMD's Zen based x86 architecture. All x86 Zen ISA and associated intrinsics are supported

-march=znver2

Use this architecture flag for enabling best code generation and tuning for AMD's Zen2 based on x86 architecture.

Code Generation Options

-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -O, -O4

Specifies which optimization level to use:

`-O0` Means "no optimization": this level compiles the fastest and generates the most debuggable code.

`-O1` Somewhere between `-O0` and `-O2`.

`-O2` Moderate level of optimization which enables most optimizations.

`-O3` Like `-O2`, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

The *-O3* level in AOCC has more optimizations when compared to the base LLVM version on which it is based. These optimizations include improved handling of indirect calls, advanced vectorization etc.

`-Ofast` Enables all the optimizations from `-O3` along with other aggressive optimizations that may violate strict compliance with language standards.

The *-Ofast* level in AOCC has more optimizations when compared to the base LLVM version on which it is based. These optimizations include partial unswitching, improvements to inlining, unrolling etc.

`-Os` Like `-O2` with extra optimizations to reduce code size.

`-Oz` Like `-Os` (and thus `-O2`), but reduces code size further.

`-O` Equivalent to `-O2`.

`-O4` and higher

Currently equivalent to `-O3`

More information on many of these options is available at http://llvm.org/docs/Passes.html.

The following optimizations are not present in LLVM and are specific to AOCC

-fstruct-layout=[1,2,3,4,5,6,7]

Analyzes the whole program to determine if the structures in the code can be peeled and if pointers in the structure can be compressed. If feasible, this optimization transforms the code to enable these improvements. This transformation is likely to improve cache utilization and memory bandwidth. This, in turn, is expected to improve the scalability of programs executed on multiple cores.

This is effective only under flto as the whole program analysis is required to perform this optimization. You can choose different levels of aggressiveness with which this optimization can be applied to your application with 1 being the least aggressive and 7 being the most aggressive level.

- fstruct-layout=1 enables structure peeling
- fstruct-layout=2 enables structure peeling and selectively compresses self-referential pointers in these structures to 32-bit pointers wherever safe
- fstruct-layout=3 enables structure peeling and selectively compresses self-referential pointers in these structures to 16-bit pointers wherever safe
- fstruct-layout=4 enables structure peeling, pointer compression as in level 2 and further enables compression of structure fields which are of integer type. This is performed under a strict safety check.

- fstruct-layout=5 enables structure peeling, pointer compression as in level 3 and further enables compression of structure fields which are of integer type. This is performed under a strict safety check.
- fstruct-layout=6 enables structure peeling, pointer compression as in level 2 and further enables compression of structure fields which are of type 64-bit 'signed int' or 'unsigned Int'. The user needs to ensure that the values assigned to 64-bit 'signed int' fields are in range -(2^31 - 1) to +(2^31 - 1) and 64-bit 'unsigned int' fields are in range 0 to +(2^31 - 1), otherwise, incorrect results may be obtained. This compression is performed without considering any safety analysis and so the user needs to ensure the safety based on the program compiled.
- fstruct-layout=7 enables structure peeling, pointer compression as in level 3 and further enables compression of structure fields which are of type 64-bit 'signed int' or 'unsigned Int'. The user needs to ensure that the values assigned to 64-bit 'signed int' fields are in range -(2^31 - 1) to +(2^31 - 1) and 64-bit 'unsigned int' field are in range 0 to +(2^31 - 1) , otherwise, incorrect results may be obtained. This compression is performed without considering any safety analysis and so the user needs to ensure the safety based on the program compiled.

Note:
fstruct-layout=4 and fstruct-layout=5 are derived from fstruct-layout=2 and fstruct-layout=3 respectively with the added feature of safe compression of integer fields in structures. Going from fstruct-layout=4 to fstruct-layout=5 may result in higher performance if the pointer values are such that the pointers can be compressed to 16-bits.

fstruct-layout=6 and fstruct-layout=7 are derived from fstruct-layout=2 and fstruct-layout=3 respectively with the added feature of compression of integer fields in structures. These are similar to fstruct-layout=4 and fstruct-layout=5, but here, the integer fields of the structures are always compressed from 64-bits to 32-bits without any safety guarantee.

-fitodcalls

Promotes indirect to direct calls by placing conditional calls. Application or benchmarks that have small and deterministic set of target functions for function pointers that are passed as call parameters benefit from this optimization. Indirect-to-direct call promotion transforms the code to use all possible determined targets under runtime checks and falls back to the original code for all other cases. Runtime checks are introduced by the compiler for each of these possible function pointer targets followed by direct calls to the targets.
This is a link time optimization which is invoked as *-flto -fitodcalls*.

-fitodcallsbyclone

Performs value specialization for functions with function pointers passed as an argument. It does this specialization by generating a clone of the function. The cloning of the function happens in the call chain as needed to allow conversion of indirect function call to direct call. This complement -fitodcalls optimization and is also a link time optimization which is invoked as *-flto -fitodcallsbyclone*.

-fremap-arrays

Transforms the data layout of a single dimensional array to provide better cache locality. This optimization is effective only under flto as the whole program analysis is required to perform this optimization which can be invoked as *-flto -fremap-arrays*.

-finline-aggressive

Enables improved inlining capability through better heuristics. This optimization is more effective when using with flto as the whole program analysis is required to perform this optimization, which can be invoked as *-flto -finline-aggressive*.

The following optimization options needs to be invoked through driver "-mllvm <options>" as mentioned in below section

-enable-partial-unswitch

Enables partial loop un-switching which is an enhancement to the existing loop un-switching optimization in LLVM. Partial loop un-switching hoists a condition inside a loop from a path for which the execution condition remains invariant whereas the original loop un-switching works for condition that is completely loop invariant. The condition inside the loop gets hoisted out from the invariant path and original loop is retained for the path where condition is variant.

-aggressive-loop-unswitch

Experimental option which enables aggressive loop unswitching heuristic (including -enable-partial-unswitch) based on the usage of the branch conditional values. Loop unswitching leads to code-bloat. Code-bloat can be minimized if the hoisted condition is executed more often. This heuristic prioritizes the conditions based on the number of times they are used within the loop. The heuristic can be controlled with the following options:

- **-unswitch-identical-branches-min-count=<n>**

  Enables unswitching of a loop with respect to a branch conditional value (B), where B appears in at least <n> compares in the loop. This option is enabled with -aggressive-loop-unswitch. Default value is 3.

  Usage: *-mllvm -aggressive-loop-unswitch -mllvm -unswitch-identical-branches-min-count=<n>*
  where n is a positive integer and lower value of <n> facilitates more unswitching

- **-unswitch-identical-branches-max-count=<n>**

  Enables unswitching of a loop with respect to a branch conditional value (B), where B appears in at most <n> compares in the loop. This option is enabled with -aggressive-loop-unswitch. Default value is 6.

  Usage: *-mllvm -aggressive-loop-unswitch -mllvm -unswitch-identical-branches-max-count=<n>*
  where n is a positive integer and higher value of <n> facilitates more unswithing

-enable-strided-vectorization

Enables strided memory vectorization as an enhancement to the interleaved vectorization framework present in LLVM. It enables effective use of gather and scatter kind of instruction patterns. This flag needs to be used along with the interleave vectorization flag.

-enable-epilog-vectorization

Enables vectorization of epilog-iterations as an enhancement to existing vectorization framework. This enables generation of an additional epilog vector loop version for the remainder iterations of the original vector loop. The vector size or factor of the original loop should be large enough to allow effective epilog vectorization of the remaining iterations. This this optimization takes effect only when the original vector loop is vectorized with a vector width or factor of sixteen. This vectorization width of sixteen may be overwritten by *-min-width-epilog-vectorization* command line option.

-enable-redundant-movs

Removes any redundant mov operations including redundant loads from memory and stores to memory. This may be invoked by *-Wl,-plugin-opt=-enable-redundant-movs*.

-merge-constant

Attempts to promote frequently occurring constants to registers. The aim is to reduce the size of the instruction encoding for instructions using constants and thereby obtain performance improvement.

-function-specialize

Optimizes functions with compile time constant formal arguments

-lv-function-specialization

Generates specialized function versions when the loops inside function are vectorizable and the arguments are not aliased with each other

-enable-vectorize-compares

Enables vectorization on certain loops with conditional breaks, assuming the memory access are safely bound within the page boundary.

-inline-recursion=[1,2,3,4]

Enables inlining for recursive functions based on heuristics with level 4 being most aggressive. Default level will be 2. Higher levels may lead to code bloat due to expansion of recursive functions at call sites.

- For level 1-2: Enables inlining for recursive functions using heuristics with inline depth 1. Level 2 uses more aggressive heuristics
- For level 3: Enables inlining for all recursive functions with inline depth 1
- For level 4: Enables inlining for all recursive function with inline depth 10

This is more effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as *-flto -inline-recursion=[1,2,3,4]*.

-reduce-array-computations=[1,2,3]

Performs array dataflow analysis and optimizes the unused array computations

- reduce-array-computations=1 Eliminates the computations on unused array elements
- reduce-array-computations=2 Eliminates the computations on zero valued array elements

- reduce-array-computations=3 Eliminates the computations on unused and zero valued array elements (combination of 1 and 2)

This optimization is effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as *-flto -reduce-array-computations=[1,2,3]*.

-global-vectorize-slp

Vectorizes the straight-line code inside a basic block with data reordering vector operations

-region-vectorize

Experimental flag for enabling vectorization on certain loops with complex control flow which the normal vectorizer cannot handle.

This optimization is effective with flto as the whole program analysis is required to perform this optimization, which can be invoked as *-flto -region-vectorize*.

Deprecated options:

vectorize-memory-aggressively - from AOCC 2.2.0

Driver Options

-mllvm <options>

Need to provide -mllvm so that the option can pass through the compiler front end and get applied on the optimizer where this optimization is implemented.

For example: -mllvm -enable-strided-vectorization