Flang - the Fortran Compiler

Contents

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 in to tokens, then the parser transforms these tokens in to Abstract Syntax Tree (AST). This AST is then transformed in to canonical form which is used to generate ILM code. Then flang2 takes up this ILM code and transforms in to 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

1. This is an early drop (alpha quality) for your experimentation
2. Do send us your feedbacks and the list of improvements you would like to see in Flang if it needs to become AOCC's primary Fortran compiler, either on support forum AMD Server Gurus or email toolchainsupport@amd.com

AOCC 1.2 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

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

-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

-Mrecursive
Generate code to allow recursive subprograms

-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

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

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`

The following optimizations are enabled at -O1 level.

-targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -ipsccp -globalopt -domtree -mem2reg -deadargelim -basicaa -aa -instcombine -simplifycfg -pgo-icall-prom -basiccg -globals-aa -prune-eh -always-inline -functionattrs -sroa -early-cse -speculative-execution -lazy-value-info -jump-threading -

correlated-propagation -libcalls-shrinkwrap -tailcallelim -reassociate -loops -loop-simplify -lcssa -scalar-evolution -loop-rotate -licm -analyze-partial-invar -loop-unswitch -indvars -loop-idiom -loop-deletion -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -peephole-loop-simplify -loop-unroll -memdep -memcpyopt -sccp -demanded-bits -bdce -dse -postdomtree -adce -barrier -rpo-functionattrs -float2int -branch-prob -block-freq -loop-load-elim -alignment-from-assumptions -strip-dead-prototypes -instsimplify

The following optimizations are added on at the -O2 level.

-vectorize-loops -vectorize-slp -itodcalls -itodcallsbyclone -inline -mldst-motion -gvn -elim-avail-extern -slpinstcombine -globaldce -constmerge -loop-sink

The following optimizations are dropped at the -O2 level.

-always-inline

The following optimizations are added on at the -O3 level.

-argpromotion

The following optimizations are added on at the -Ofast level.

-aggressive-loop-unswitch -enable-nans-for-sqrt -menable-no-infs -menable-no-nans -menable-unsafe-fp-math -fno-signed-zeros -freciprocal-math -fno-trapping-math -ffp-contract=fast -ffast-math

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]

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 5 being the most aggressive level.

- fstruct-layout=1 enables structure peeling

- fstruct-layout=2 enables structure peeling and pointer compression when size fits within 64KB and 4GB.
- fstruct-layout=3 enables structure peeling and pointer compression when size fits within 64KB.
- fstruct-layout=4 enables data compression in addition to level 2
- fstruct-layout=5 enables data compression in addition to level 3

-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.

-loop-unswitch-aggressive

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 atleast <n> compares in the loop. This option is enabled with -loop-unswitch-aggressive. Default value is 3.

Usage: *-mllvm -loop-unswitch-aggressive -mllvm -unswitch-identical-branches-min-count=<n> where n is a positive integer*

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

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

Usage: *-mllvm -loop-unswitch-aggressive -mllvm -unswitch-identical-branches-max-count=<n> where n is a positive integer*

-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.

-vectorize-memory-aggressively

This makes an assumption that memory accesses do not alias in the process of vectorizing a loop. The loop vectorizer generates runtime checks for all unique memory accesses when the compiler is not sure about memory aliasing. The result of the runtime check determines whether the vectorized loop version or the scalar loop version is executed. If the runtime check detects any memory aliasing, then the scalar loop is executed otherwise, the vector loop executed. This option forces the loop vectorizer not to generate these runtime alias checks by making an assumption that memory accesses do not alias. The responsibility of correct usage of this option is left to the user. This option may be used only if memory accesses do not overlap in loops.

-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

-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 heurstics 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]*.

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