Clang - the C, C++ Compiler

Contents

SYNOPSIS

clang [*options*] *filename ...*

DESCRIPTION

clang is a C, C++, and Objective-C compiler which encompasses preprocessing, parsing, optimization, code generation, assembly, and linking. Depending on which high-level mode setting is passed, Clang will stop before doing a full link. While Clang is highly integrated, it is important to understand the stages of compilation, to understand how to invoke it. These stages are:

Driver
: The clang executable is actually a small driver which controls the overall execution of other tools such as the compiler, assembler and linker. Typically you do not need to interact with the driver, but you transparently use it to run the other tools.

Preprocessing
: This stage handles tokenization of the input source file, macro expansion, #include expansion and handling of other preprocessor directives. The output of this stage is typically called a ”.i” (for C), ”.ii” (for C++), ”.mi” (for Objective-C), or ”.mii” (for Objective-C++) file.

Parsing and Semantic Analysis
: This stage parses the input file, translating preprocessor tokens into a parse tree. Once in the form of a parse tree, it applies semantic analysis to compute types for expressions as well and determine whether the code is well formed. This stage is responsible for generating most of the compiler warnings as well as parse errors. The output of this stage is an “Abstract Syntax Tree” (AST).

Code Generation and Optimization

This stage translates an AST into low-level intermediate code (known as “LLVM IR”) and ultimately to machine code. This phase is responsible for optimizing the generated code and handling target-specific code generation. The output of this stage is typically called a ”.s” file or “assembly” file.

Clang also supports the use of an integrated assembler, in which the code generator produces object files directly. This avoids the overhead of generating the ”.s” file and of calling the target assembler.

Assembler
: This stage runs the target assembler to translate the output of the compiler into a target object file. The output of this stage is typically called a ”.o” file or “object” file.

Linker
: This stage runs the target linker to merge multiple object files into an executable or dynamic library. The output of this stage is typically called an “a.out”, ”.dylib” or ”.so” file.

Support for Annex F (IEEE-754 / IEC 559) of C99/C11

The Clang compiler does not support IEC 559 math functionality. Clang also does not control and honour the definition of __STDC_IEC_559__ macro. Under specific options such as -Ofast and -ffast-math, the compiler will enable a range of optimizations that provide faster mathematical operations that may not conform to the IEEE-754 specifications. The macro __STDC_IEC_559__ value may be defined but ignored when these faster optimizations are enabled.

OPTIONS

Target Selection Options

-march`=<cpu>`
: Specify that Clang 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

  The current release has

- Improved code generation under “znver1” for AMD’s Zen based X86 architecture
- Improved register allocation
- Improved packing for vector instructions

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 4.0 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.

-Ofast by default enables use of –ffast-math option. Use of –ffast-math, either implicitly through –Ofast or explicitly specifying it with other optimization levels like –O3 etc, enables a range of optimizations that provide faster, though sometimes less precise, mathematical operations that may not conform to the IEEE-754 specifications. When this option is specified, __STDC_IEC_559__ macro is ignored even if set by the system headers

The *-Ofast* level in AOCC has more optimizations when compared to the base LLVM 4.0 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.

-remove-dead-loops -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.

`-ffast-math`

Enables a range of optimizations that provide faster, though sometimes less precise, mathematical operations that may not conform to the IEEE-754 specifications. When this option is specified, __STDC_IEC_559__ macro is ignored even if set by the system headers. The option is enabled under Ofast by default.

The following optimizations are not present in LLVM 4.0 and are specific to AOCC.

-fstruct-layout=[1,2,3]

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.

The option 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 3 being the most aggressive level Use -fstruct-layout=3 when you know the allocated size of array of structures fits within 64KB. Use the value of 2 when a similar size exceeds 64KB but does not exceed 4GB.

-fitodcalls

This optimization 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

This optimization does 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 complements -fitodcalls optimization and is also a link time optimization which is invoked as *-flto -fitodcallsbyclone*.

-enable-partial-unswitch

This optimization enables partial loop un-switching which is an enhancement to the existing loop un-switching optimization in LLVM 4.0. 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 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.

-disable-vect-cmp

Certain loops with conditional breaks maybe vectorized by default at O2 and above. In some extreme situations this may result in unsafe behavior. Use this option to disable vectorization of such loops.

-remove-dead-loops

This is an optimization which removes dead loops without caring about loop bounds. It enables elimination of dead loop whose bounds are unknown.

-loop-unswitch-aggressive

Enables aggressive loop unswitching heuristic based on the usage of the branch conditional values.

-enable-strided-vectorization

This optimization enables strided memory vectorization as an enhancement to the interleaved vectorization framework present in LLVM 4.0. 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

This optimization 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 optimization 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.

-finline-aggressive

This option enables improved inlining capability through better heuristics. This is a link time optimization which may be invoked by *-flto -finline-aggressive*.

-fremap-arrays

This optimization transforms the data layout of a single dimensional array to provide better cache locality. This is a link time optimization and is invoked by *-flto -fremap-arrays*.

-enable-redundant-movs

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

-merge-constant

This optimization 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.

Driver Options

-mllvm `-enable-strided-vectorization`

This optimization is not present in LLVM 4.0 and is specific to AOCC.

This optimization involves strided memory vectorization as an enhancement to LLVM's interleaved vectorization framework. This enables effective use of gather and scatter kind of data access patterns.

You need to provide -mllvm so that the option can pass through the compiler frontend and get applied on the optimizer where this optimization is implemented.

Note: For more information on Clang options do refer to Clang-CommandGuide