DragonEgg - the Fortran compiler

Contents

SYNOPSIS

gfortran [*gfortran opt flags*] -fplugin=<absolute path to AOCC install bin directory>dragonegg.so -fplugin-arg-dragonegg-llvm-option="[*AOCC optimization flags*]" filename ...

DESCRIPTION

DragonEgg is a gcc plugin that replaces GCC's optimizers and code generators with those from the LLVM project. DragonEgg that comes with AOCC works with gcc-4.8.x, has been tested for x86-32/x86-64 targets and has been successfully used on various Linux platforms. DragonEgg is the compiler to be used for Fortran programs.

GFortran is the actual frontend for Fortran programs responsible for preprocessing, parsing and semantic analysis generating the GCC GIMPLE intermediate representation (IR). DragonEgg is a GNU plugin, plugging into GFortran compilation flow. It implements the GNU plugin API. With the plugin architecture, DragonEgg becomes the compiler driver, driving the different phases of compilation. DragonEgg is responsible for transforming GIMPLE IR to LLVM IR. DragonEgg driver transfers this transformed LLVM IR to LLVM optimizer for optimization and target code generation.

IEEE-754 Support

The dragonegg 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

DragonEgg driver relies on AOCC's optimizer and code generator to transform the available LLVM IR (derived from GIMPLE IR) and generate the best code for the target x86 platform.

BUILD DRAGONEGG PLUGIN

Prerequisites

GCC version to be used: gcc-4.8.x (can be downloaded from https://ftp.gnu.org/gnu/gcc/)

Steps to build dragonegg plugin

- DragonEgg sources are available under AOCC-1.0-FortranPlugin.tar pacakge
- Extract the tar package, i.e., *tar -xf AOCC-1.0-FortranPlugin.tar*
- cd into dragonegg directory
- Command line to build dragonegg:

GCC=/Path to gcc-4.8.x binaries/gcc LLVM_CONFIG=/Path to llvm binaries/llvm-config ENABLE_AMD_LLVM=1 make

ENABLE_AMD_LLVM=1 enables the AMD LLVM specific code in dragonegg.

This will generate shared library file *dragonegg.so*

USAGE OF DRAGONEGG PLUGIN TO COMPILE FORTRAN PROGRAM

gfortran [*gfortran opt flags*] [*gfortran option to enable specific llvm optimization*] -fplugin=<absolute path to AOCC install bin directory>dragonegg.so -fplugin-arg-dragonegg-llvm-option="[*AOCC optimization flags*]" filename ...

Gfortran Options

gfortran opt flags: Since we are using GFortran, only as a frontend, it is recommended to use an out of the box GFortran optimization like *-O2* or *-O3* (if needed *-Ofast*) and pass all optimization flags to AOCC.

gfortran option to enable specific llvm optimization: These options are hint to enable specific llvm optimizations. Options which are used are:

-funroll-loops, -fdefault-integer-8, -ffast-math

Target Selection Options

-march=<cpu>, **-mavx**

Specifes that DragonEgg/AOCC 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.

Note: Zen Target Selection Option: **-madx**

Use this architecture flag for enabling highly tuned 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"
- Improved register allocation
- Improved packing for vector instructions

Code Generation and LLVM IR Options

-fplugin-arg-dragonegg-llvm-ir-optimize=N

Run the LLVM IR optimizers at optimization level N, overriding the GCC optimization level. Usually if you pass -O1, -O2 etc to GCC then the LLVM IR level optimizers are also run at -O1, -O2 etc. Use this option to change this, disassociating the LLVM optimization level from the GCC one. For example, -fplugin-arg-dragonegg-llvm-ir-optimize=0 disables all LLVM IR optimizations.

-fplugin-arg-dragonegg-llvm-codegen-optimize=N

Run the LLVM code generator optimizers at optimization level N, overriding the GCC optimization level. Usually if you pass -O1, -O2 etc to GCC then the LLVM code generators optimize at a corresponding level. Use this option to change this, disassociating the LLVM optimization level from the GCC one.

Various optimization that can be used are:

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

Specify 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`

LLVM Optimizations enabled under different optimization levels:

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 -inline -mldst-motion -gvn -elim-avail-extern -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.

-unroll-aggressive

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

AOCC optimization flags

*-fplugin-arg-dragonegg-llvm-option="[*AOCC optimization flags*]"*

Used to pass command line optimization through to AOCC. If you want to pass an option that contains equals signs then you need to use colons (':') instead of '='.

For example: *-fplugin-arg-dragonegg-llvm-option=" -enable-iv-split, -branch-combine, -unroll-threshold:150"*

The following optimizations are specific to AOCC and are not present in LLVM 4.0. They have been implemented for Fortran programs.

`-enable-iv-split`

Enables splitting of long live rages of loop induction variables which span loop boundaries. This helps reduce register pressure and can help avoid needless spills to memory and reloads from memory.

`-lsr-in-nested-loop`

Loop strength reduction (lsr) is used to simplify address calculations which address array elements inside a loop. This optimization extends the applicability of lsr optimization in nested loops whereas in LLVM 4.0 it works only for the innermost loop.