GCC/Clang Optimizations for Embedded Linux

Khem Raj, Comcast

Embedded Linux Conference & OpenIOT summit Portland, OR

Introduction To Clang

- Native compiler FrontEnd to LLVM Infrastructure
- Supports C/C++ and Objective-C
- The LLVM Project is a collection of modular and reusable compiler and toolchain technologies - llvm.org
- First release in 2003
- Latest Release 3.9.1 (Dec 2016)
- Pronounced as /klaNG/

GCC

- The GNU compiler Collection
- Native and cross compiler
- Multiple language frontends
  - C/C++/Fortran/Ada/Golang...
- Modular design
- Supports multiple architectures
  - List of Supported Backends
- Latest stable release (6.3) Dec 2016

On Optimization Flags

- O0 - Unoptimized (faster compile time)
- O1 - General optimizations no speed/size tradeoff
- O2 - More aggressive size and speed optimization
- O3 - Favor speed over size
- Os
  - Like O2 but does not enable opt passed which increase size
- Ofast
  - O3 plus inaccurate math
- Og
  - Optimize for debugging experience
- Clang has -Oz which optimizes for size more aggressively
  - Disables loop vectorization
- -O is equal to -O2 in Clang and -O1 in gcc

Optimization Flags

- -O<n> are bundle of individual optimization passes
    - GCC can dump the collection used

```
gcc -O2 -fverbose-asm -S mem.c
```

```
# GNU C11 (GCC) version 6.3.1 20170109 (x86_64-pc-linux-gnu)
#       compiled by GNU C version 6.3.1 20170109, GMP version 6.1.2, MPFR version 3.1.5-p2,
MPC version 1.0.3, isl version 0.15
# GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
# options passed:  mem.c -mtune=generic -march=x86-64 -O2 -fverbose-asm
# options enabled:  -faggressive-loop-optimizations -falign-labels
# -fasynchronous-unwind-tables -fauto-inc-dec -fbranch-count-reg
# -fcaller-saves -fchkp-check-incomplete-type -fchkp-check-read
# -fchkp-check-write -fchkp-instrument-calls -fchkp-narrow-bounds
# -fchkp-optimize -fchkp-store-bounds -fchkp-use-static-bounds
# -fchkp-use-static-const-bounds -fchkp-use-wrappers
# -fcombine-stack-adjustments -fcommon -fcompare-elim -fcprop-registers
# -fcrossjumping -fcse-follow-jumps -fdefer-pop
# -fdelete-null-pointer-checks -fdevirtualize -fdevirtualize-speculatively
# -fdwarf2-cfi-asm -fearly-inlining -feliminate-unused-debug-types
# -fexpensive-optimizations -fforward-propagate -ffunction-cse -fgcse
# -fgcse-lm -fgnu-runtime -fgnu-unique -fguess-branch-probability
```

```
gcc -c -Q -O2 --help=optimizers
```

Optimization Flags

- Any optimization pass can be enabled/disabled individually
  - Sould be rightmost on cmdline to be effective

```
gcc -O2 -fno-aggressive-loop-optimizations -fverbose-asm mem.c -S
```

```
# GNU C11 (GCC) version 6.3.1 20170109 (x86_64-pc-linux-gnu)
#       compiled by GNU C version 6.3.1 20170109, GMP version 6.1.2, MPFR version
3.1.5-p2, MPC version 1.0.3, isl version 0.15
# GGC heuristics: --param ggc-min-expand=100 --param ggc-min-heapsize=131072
# options passed:  mem.c -mtune=generic -march=x86-64 -auxbase-strip mem.s
# -O2 -fno-aggressive-loop-optimizations -fverbose-asm
# options enabled:  -falign-labels -fasynchronous-unwind-tables
# -fauto-inc-dec -fbranch-count-reg -fcaller-saves
# -fchkp-check-incomplete-type -fchkp-check-read -fchkp-check-write
# -fchkp-instrument-calls -fchkp-narrow-bounds -fchkp-optimize
# -fchkp-store-bounds -fchkp-use-static-bounds
# -fchkp-use-static-const-bounds -fchkp-use-wrappers
# -fcombine-stack-adjustments -fcommon -fcompare-elim -fcprop-registers
# -fcrossjumping -fcse-follow-jumps -fdefer-pop
# -fdelete-null-pointer-checks -fdevirtualize -fdevirtualize-speculatively
….
```

- https://gcc.gnu.org/onlinedocs/gcc-6.3.0/gcc/Optimize-Options.html

Aliasing

- Instruction compiler on disabling/enabling
  - -fstrict-aliasing
  - Enabled at -O2 by default
- Use -Wstrict-aliasing for finding violations
- Use “restrict” keyword
- Pointer to int and pointer to long may not alias
- Type conversion may break compiler’s assumptions
- Uint8_t and int8_t behaved differently then char
  - Fixed with gcc 6.0

```
int func(int *x, int *y) {
  *x = 100;
  *y = -100;
  return *x;
}
```

```
int func(int *x, long *y) {
  *x = 100;
  *y = 1000;
  return *x;
}
```

Inlining

- Use ‘inline’ keyword
  - Hints to compiler for considering the function for inlining
- Force inlining
  - Use ‘always_inline’ function attribute

```
inline void foo (const char)
__attribute__((always_inline));
```

- GCC has 3 semantic implementations
  - Gnu89 inline
  - C99 inline
  - C++ inline

Stack Optimizations

- Determine static stack usage
  - -fstack-usage
    - Not available with clang
  - Information is in .su file

```
mem.c:6:5:main   48   static
```

- What contributes towards stack size
  - Local vars
  - Temporary data
  - Function parameters
  - Return addresses

Stack Optimizations

- Know your local variables
  - Avoid large stack
  - Use data in-place instead of copying
  - Use inline functions
- Avoid Recursive functions
- Limit function call-chains
- Use --Wstack-usage

```
gcc -Wstack-usage=10  mem.c
mem.c:6:5: warning: stack usage is 48 bytes [-Wstack-usage=]
 int main(int argc, char *argv[])
```

- -fconserve-stack
  - Minimize stack might run slower

Size optimizations

- Adjust Stack alignment
  - -mpreferred-stack-boundary=n
  - Clang has -mstack-alignment=<value>
- Identical constants can be merged to save space
  - Gcc has -fmerge-constants
  - Clang has -fmerge-all-constants
- -fomit-frame-pointer
  - Debugging will suffer
- -ffunction-sections
  - Put each global or static function in its own section named .text.<name>
- -fdata-sections
  - Put each global or static variable into .data.variable_name, .rodata.variable_name or .bss.variable_name.

Profile Guided Optimization

- Help compiler find optimized execution path
- Statistical
  - Imprecise
  - Low Overhead
- Instrumented
  - Precise
  - Intrusive

Feedback-Directed Optimization

- Uses static instrumentation for data collection
  - Build instrumented code (-fprofile-generate)
  - Run instrumented code with training data
    - Quite slow due to overhead
  - Build optimized version of code by using execution profile data
    - -fprofile-use=<name of execution profile file>
- Hard to use due to
  - high overhead of instrumented run
  - Difficulties in generating training data
  - Dual compile is tedious
- Use AutoFDO (https://github.com/google/autofdo)
  - Uses perf and uses sampling based profile

Link Time Optimization - Clang

- Inter-modular optimizations at link time

```
clang -c -emit-llvm mem.c -o mem.o - Generates bitcode
Clang -c main.c -o main.o
Clang -o main main.o mem.o
```

- libLTO to handle llvm bitcode
- -flto
    - full (default)
    - thin (ThinLTO)
        - Faster compile time with similar gains
        - Needs gold linker

Link Time Optimization - GCC

- -flto

```
gcc -c -flto mem.c -o mem.o - Generates gimple bitcode
gcc -c -flto main.c -o main.o
gcc -flto -o main main.o mem.o
```

- Archives with LTO can be generated using gcc-ar and gcc-ranlib
- -fuse-linker-plugin - Needed during link to understand .a with LTO
    - Needs linker with plugin support
- -flto -ffat-lto-objects
    - Makes code suitable for both LTO and non-LTO linking
- Combining lto with -g might not work as expected

Loop Optimizations - Auto-Vectorization

- Will try to make use of SIMD instructions
- Enabled at -O3
- GCC
  - -ftree-vectorize to enable it explicitly
    - Need SIMD options enabled e.g. -maltivec/ppc, -msseX/x86
- Clang
  - Disable with -fno-vectorize
  - -force-vector-width=<n>, n controls the SIMD width
  - Pragma hints
    - `#pragma clang loop vectorize(enable) interleave(enable)`
- Both compilers support SLP vectorization (a.k.a. superword-level parallelism)
  - Clang seems to have second phase as well
    - `-fslp-vectorize-aggressive`

Target Specific Optimizations

- CPU Type
  - -march - Select Instructions set for assembly generation
  - -mtune - Target processor for tuning performance
  - -mcpu - Target processor with feature modifiers
- SIMD
  - ARM/Neon, x86/SSE..
- Target ABIs
  - mips/-mplt
- Explore target specific optimizations
  - gcc --target-help
- Machine specific optimizations
  - https://gcc.gnu.org/onlinedocs/gcc/Submodel-Options.html#Submodel-Options

Built-in Functions

- Many targets support built-ins specific to those machines
  - Compiler can schedule those function calls.
  - GCC
    - https://gcc.gnu.org/onlinedocs/gcc/Target-Builtins.html#Target-Builtins
  - Clang
    - https://clang.llvm.org/docs/LanguageExtensions.html#builtin-functions

Unsupported GCC extensions in Clang

- Variable length arrays in structs
- __builtin-apply
- Nested functions
- __builtin_va_arg_pack/__builtin_va_arg_pack_len
- Forward-declaration of function parameters
  - Use -std=c99 -pedantic for consistent behavior

```
void func(int i; int array[i])
{
}
```

Summary

- Help the compiler, in turn it will help you
- Evaluate the impact of optimizations
  - Every load is not same
- Know your architecture, cache sizes, instruction latencies
- Profile your code before optimizing
  - Data is truth
- Writing portable code is a good habit
- Over optimization
  - Apply your judgement