Compiler feedback and optimisation

Performance Analysis Workshop Series 2024

Pawel Radtke
Durham, 9 April 2024

Intro

- Compiler landscape
- Common functional options
- Machine-specific options
- Translation-unit optimisations
- Whole-program optimisations
- Speeding up compilation
- Questions

Compiler landscape

Vendor-agnostic	Vendor-specific
GNU Compiler Collection (GCC)	ARM-GCC
Clang/LLVM	Intel oneAPI ® Compiler* AOCC …
	Intel “Legacy” Compiler

* has both the LLVM and the proprietary Intel optimisers

Compiler landscape

Key differences:

- Support for cutting-edge language features
- Quality of error messages
- Cross-compatibility of invocation flags
- Code analysis tool support (e.g. Intel VTune, Intel Advisor)
- Vendor-specific optimisations
- Performance

Compiler functional options

Language level

Newest compilers support portions of C++20/23/26 but do not enable support by default.

Mismatch between compiler language level and source code language level rarely identified in error messages.

Controlled by: `-std=c++(14/17/20/23/26)`

Compiler functional options

Language level - example: Clang/LLVM 17.0.6 -std=c++17

```cpp
#include <concepts>

template<typename T>
concept Sized = requires(T a)
{
   { a.size() } -> std::same_as<int>;
};
```

```
<source>:4:1: error: unknown type name 'concept'
concept Sized = requires(T a)
^
<source>:4:26: error: 'T' does not refer to a value
concept Sized = requires(T a)
                         ^
<source>:3:19: note: declared here
template<typename T>
                  ^
2 errors generated.
Compiler returned: 1
```

Compiler functional options

Language level - example: Clang/LLVM 17.0.6 -std=c++20

```
#include <concepts>

template<typename T>
concept Sized = requires(T a)
{
   { a.size() } -> std::same_as<int>;
};
```

```
Compiler returned: 0
```

Compiler functional options

Warnings

- Help spot errors quickly
- Quality and verbosity varies across compilers
- Most compilers offer fine-grained control over reported errors
- Flag format: `-W(error-type)` `-W-no-(error-type)`

Compiler functional options

Warnings

- Most-used flags

`-Wall`	enables all warnings
`-Wextra`	enables extra warnings
`-Wpedantic`	warns about non-ISO language features
`-Werror`	turns warnings into errors

Compiler functional options

Warnings - example: Clang/LLVM 17.0.6

```
int doWork(int a) {
   int A;
   if (A < 0) {
       A = a;
   }

   return A;
}
```

```
Compiler returned: 0
```

Compiler functional options

Warnings - example: Clang/LLVM 17.0.6 -Wall -Werror

```
int doWork(int a) {
   int A;
   if (A < 0) {
       A = a;
   }

   return A;
}
```

```
<source>:3:9: error: variable 'A' is uninitialized
when used here [-Werror,-Wuninitialized]
   if (A < 0) {
       ^
<source>:2:10: note: initialize the variable 'A' to
silence this warning
   int A;
        ^
         = 0
1 error generated.
Compiler returned: 1
```

Compiler functional options

Debug levels

- Adds debugging information to the binary regardless of optimisation level
- The more aggressive optimisation, the poorer the debug info
- Controlled by `-g`
- Different levels exist, e.g. `-g0, -g1, -g2, -g3`
- Default is level 2
- Level 3 adds more info about expanded macros, might break debugger
- Special opt level `-Og` recommended for debug builds

Compiler functional options

Debug formats

- The standard DWARF format has 5 versions (current version: 5)
- Some older analysis tools are not compatible with the latest versions
- Control DWARF format version by `-gdwarf-(1-5)`

Compiler functional options

Summary

```
clang++ -std=c++20 -Og -gdwarf-4 -Wall -Werror [...]
```

Translation-unit optimisations

General concepts

- Translation unit is generally every .c or .cpp file in a project

- Optimisations are (generally) enabled, not forced
- The compiler makes the final decision
- Feedback available for many successful/unsuccessful optimisations
- Fine-grained controls are very compiler-specific

Translation-unit optimisations

Machine-specific options

- X86-64 is a “family” of architectures
- Many extensions exist, new are constantly added (e.g. AVX512 , AVX10)
- Check by `lscpu | grep Flags`

```
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc
rep_good nopl nonstop_tsc cpuid extd_apicid aperfmperf rapl pni pclmulqdq monitor ssse3 fma cx16
sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand lahf_lm cmp_legacy svm extapic
cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw ibs skinit wdt tce topoext perfctr_core
perfctr_nb bpext perfctr_llc mwaitx cpb cat_l3 cdp_l3 hw_pstate ssbd mba ibrs ibpb stibp vmmcall
fsgsbase bmi1 avx2 smep bmi2 cqm rdt_a rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec
xgetbv1 cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local clzero irperf xsaveerptr rdpru wbnoinvd
cppc arat npt lbrv svm_lock nrip_save tsc_scale vmcb_clean flushbyasid decodeassists pausefilter
pfthreshold avic v_vmsave_vmload vgif v_spec_ctrl umip rdpid overflow_recov succor smca
```

Translation-unit optimisations

Machine-specific options

- Controlled by `-march=ARCH -mtune=ARCH`
- `-march=ARCH` enables specific microarchitecture, e.g. `zen4` for latest AMD CPUs
- `-mtune=ARCH` enables specific cost model, usually the same as `-march`
- within X86-64, cross compilation between microarchitectures supported out of the box
    - “Compile on login node, run on compute node” is possible without any performance penalties
- Special value `native` specifies the host machine family
- Fine-grained control over enabled extensions allowed by `-m(extension-name)`

Translation-unit optimisations

Optimisation flags

- The gateway to optimisation
- Options:
  - `-O0` (no perf. opt.) to `-O3` (most aggressive perf. opt.)
  - `-Os` (level 2 perf. opt. and size opt.) , `-Oz` (more aggressive size opt.)
  - `-Ofast` (most aggressive + enables `-ffast-math`)
- `-ffast-math` assumes floating-point ops are commutative, often results in slightly different floating point results. Might break tests!
- `-Og` (very light optimisation + adds debug info) recommended for debug builds

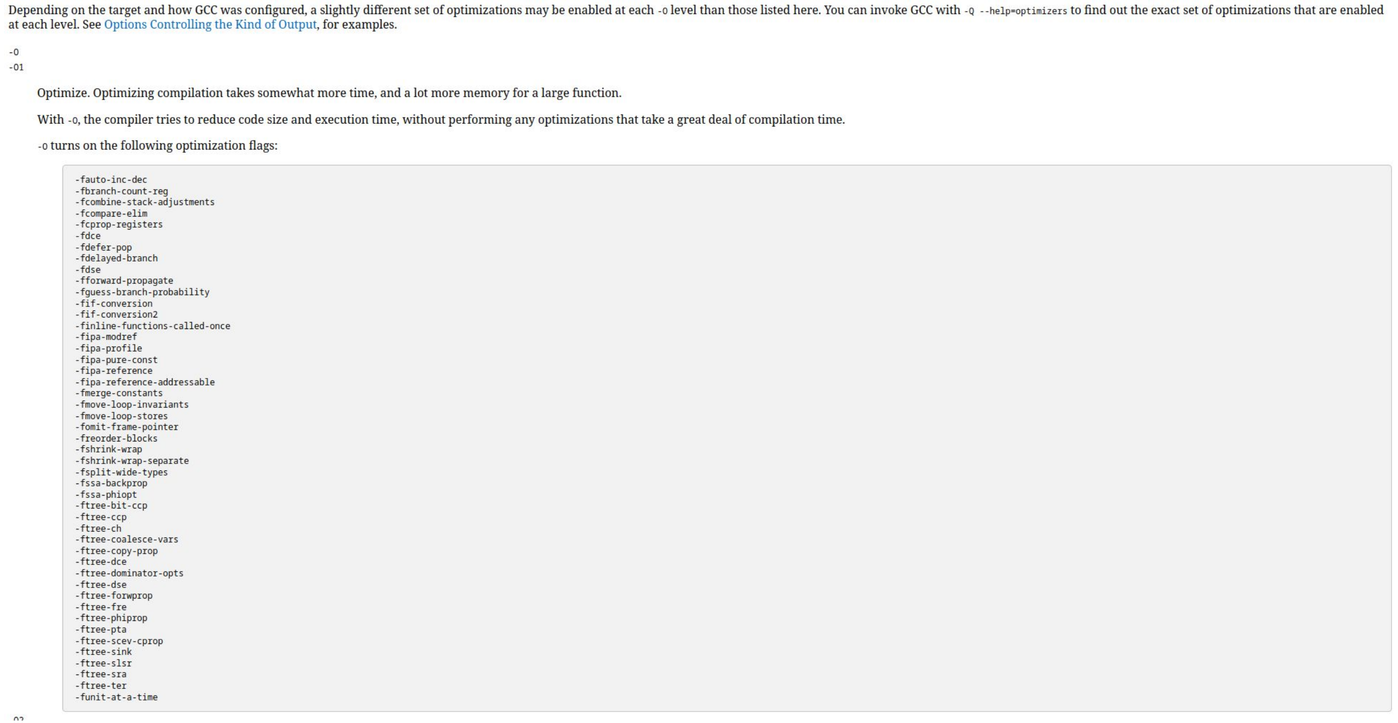
Translation-unit optimisations

Optimisation flags - under the hood

- Different levels enable sets of small optimisations
- Not guaranteed to be the same across vendors
- Level of documentation varies
- See example at GCC Docs

Translation-unit optimisations

Optimisation flags - under the hood



Depending on the target and how GCC was configured, a slightly different set of optimizations may be enabled at each `-O` level than those listed here. You can invoke GCC with `-Q --help=optimizers` to find out the exact set of optimizations that are enabled at each level. See Options Controlling the Kind of Output, for examples.

`-O`
`-O1`

Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

With `-O`, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

`-O` turns on the following optimization flags:

```
-fauto-inc-dec
-fbranch-count-reg
-fcombine-stack-adjustments
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fforward-propagate
-fguess-branch-probability
-fif-conversion
-fif-conversion2
-finline-functions-called-once
-fipa-modref
-fipa-profile
-fipa-pure-const
-fipa-reference
-fipa-reference-addressable
-fmerge-constants
-fmove-loop-invariants
-fmove-loop-stores
-fomit-frame-pointer
-freorder-blocks
-fshrink-wrap
-fshrink-wrap-separate
-fsplit-wide-types
-fssa-backprop
-fssa-phiopt
-ftree-bit-ccp
-ftree-ccp
-ftree-ch
-ftree-coalesce-vars
-ftree-copy-prop
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-pta
-ftree-scev-cprop
-ftree-sink
-ftree-slsr
-ftree-sra
-ftree-ter
-funit-at-a-time
```

Translation-unit optimisations

Optimisation flags

Many important optimisations (e.g. autovectorisation) depend both on the microarchitecture and the optimisation level!

Translation-unit optimisations

Summary

```
clang++ -march=native -mtune=native -Ofast -g [...]
```

Translation-unit optimisations

Inspecting applied optimisations

- Failed optimisations (generally) don’t generate warnings/errors
- Some optimisation passes can generate reports
- The format, content, and quality of these reports is vary highly

Translation-unit optimisations

Inspecting applied optimisations - autovectorisation

- List successful opts by `-Rpass=loop-vectorize`
- List failed opts by `-Rpass-missed=loop-vectorize`
- Explain failures by adding `-Rpass-analysis=loop-vectorize`

Translation-unit optimisations

Inspecting autovectorisation - example Clang/LLVM 17.0.6 -Rpass=loop-vectorize

```
void fma(double *a, double *b, int size) {
   for (int i = 0; i < size; i++) {
       a[i] += a[i] * b[i];
   }
}
```

```
<source>:2:5: remark: vectorized loop
(vectorization width: 8, interleaved count: 4)
[-Rpass=loop-vectorize]
    2 |     for (int i = 0; i < size; i++) {
      |     ^
Compiler returned: 0
```

Translation-unit optimisations

Inspecting autovectorisation - example Clang/LLVM 17.0.6 -O3

-Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize

```
void fmaSum(double *a, double *b, int size)
{
   for (int i = 0; i < size; i++) {
       a[i+1] += a[i] * b[i];
   }
}
```

```
<source>:9:19: remark: loop not vectorized: value
that could not be identified as reduction is used
outside the loop [-Rpass-analysis=loop-vectorize]
    9 |          a[i+1] += a[i] * b[i];
      |                    ^
<source>:8:5: remark: loop not vectorized
[-Rpass-missed=loop-vectorize]
    8 |     for (int i = 0; i < size; i++) {
      |     ^
Compiler returned: 0
```

Translation-unit optimisations

Inspecting autovectorisation

- Analysis messages can be cryptic
- Enabling autovectorisation can be as much art as engineering for non-trivial code bases
- More specialised tools exist to guide the process (e.g. MAQAO - to be covered in an upcoming session)

Translation-unit optimisations

Inspecting assembly

- Advanced method
- Can be done directly or via 3rd party tools (e.g. Intel Advisor, Godbolt.org)
- Output assembly by specifying `-S`

Translation-unit optimisations

Inspecting assembly - example Clang/LLVM 17.0.6 -O3

```c
int fib1(int a) {
   if (a <= 1) return a;
   else return fib1(a-1) + fib1(a-2);
}

int main() {
   return fib1(12);
}
```

```asm
fib1(int):
[...]
        call    fib1(int)
[...]
        ret
main:                           # @main
        mov     edi, 12
        jmp     fib1(int)       # TAILCALL
```

Translation-unit optimisations

Inspecting assembly - example Clang/LLVM 17.0.6 -O3

```
constexpr int fib2(int n) {
      int tmp = 0;
      int a = 1;
      int b = 0;
      for (int i = 0; i < n; i++) {
          tmp = a;
          a = a + b;
          b = tmp;
      }
      return b;
}
int main() {
   return fib1(12);
}
```

```
main:                                   # @main
        mov     eax, 144
        ret
```

Whole-program optimisations

General concepts

- Work on whole programs
    - link time - interprocedural optimisation (IPO) , also known as link-time optimisation (LTO)
    - re-compile based on a runtime profile - profile-guided optimisation (PGO)
- Coarse-grained control (on/off)

Whole-program optimisations

Link-time optimisation

- Out of the box, many opts (e.g. inlining, autovec.) work only within a translation unit
- Some opts can be reintroduced across translation units by including source information in the object files (usually in an intermediate format such as GIMPLE or LLVM IR)
- Enabled by `-flto` both at compile time and link time

Whole-program optimisations

Link-time optimisation - Jumbo builds

- Some build systems (e.g. CMake) support automatic concatenation of all source files into a single translation unit
- This slows down compilation but enables more optimisation otherwise delegated to LTO
- The concept is often referred to as a “Jumbo build” or a “Unity build”

Whole-program optimisations

Profile-guided optimisation

- Some aspects of the generated code
  - Code memory layout
  - Jumps
  - [...]

  Can be optimised based on statistical data from runtime.

- The process of gathering runtime data and incorporating it into an executable is called Profile-guided optimisation.

Whole-program optimisations

Profile-guided optimisation

- Compile code with data-gathering extensions `-O3 [...] -fprofile-generate`
- Run application (profile gets saved in the working directory automatically)
- Post-process profile data (compiler-dependent)
- Recompile code with application profile `-O3 [...] -fprofile-use`
- Validate results

Whole-program optimisations

Position-dependent code

- By default compilers generate position-independent code
- This adds a level of indirection to functions and data through a Global Offset Table (GOT)
- Executable/statically linked code often can be compiled to a faster, position-dependent form
- Enabled by `-fno-PIC`

Speeding up compilation

General tips

- Use a high-performance linker, e.g. `-fuse-ld=mold`
- Enable multithreaded builds `make -jXX`
- Use `-Og` for test builds
- Use pre-compiled headers - build-system specific, supported by CMake
- (anecdotal) get *faster* RAM

Thank you

Questions