TECHNISCHE
UNIVERSITÄT
DRESDEN

Center for Information Services and High Performance Computing (ZIH)

Compiler Options

Linux/x86 Performance Practical, 17.06.2009

Zellescher Weg 12

Willers-Bau A106

Tel. +49 351 - 463 - 31945

Ulf Markwardt (ulf.markwardt@tu-dresden.de)

Matthias Lieber (matthias.lieber@tu-dresden.de)

Center for Information Services &
High Performance Computing

General Optimization

- General flags imply many optimizations with a simple flag
- **`-O0`** – no optimization at all, fastest compilation, GNU default
- **`-O1`** – minimize code size with small speed optimizations
- **`-O2`** – maximize program speed, Intel default
- **`-O3`** – more aggressive optimizations than **`-O2`**, but not always better
- Specific meaning of the flags in not the same between different compilers
  - e.g. Intel **`-O2`** includes function inlining, while GNU does not
  - Read compilers manual/manpage
- All optimization levels except **`-O0`** may affect debugging (**`-g`**)
  - e.g. optimizing functions/variables away, reordering of statements
- But debugging does (practically) not affect optimization

Specific Optimization Flags

- Compilers offer tons of specific optimization flags
- Not compatible across compilers
- Address specific optimization strategies
  - May or may not increase execution speed
  - May sometimes even slow your program down
- Include straightforward, harmless optimizations but also aggressive strategies

Inlining

- Inlining replaces the call to a function by the function’s code
- Reduces function call overhead for small, often called functions
- Compiler knows context of the specific function call, which allows further optimizations, e.g. propagation of constants
- Good for object-oriented code (lots of small functions)
- Only works within a single source file
- Enable function inlining: **`-finline-functions`**
  - Intel: **`-O2`**, **`-O3`** imply inlining
  - Intel: **`-ip`** implies inlining and additional interprocedural optimizations
  - GNU: **`-O3`** implies inlining
- Control the max. size of functions that can be inlined:
  - Intel: **`-inline-factor`**
  - GNU: **`-finline-limit`**

Aliasing

- Aliasing means, that a memory address can be accessed by different symbolic names (variables, pointers)
- Aliasing prohibits optimizations, e.g.:
  - Compiler could propagate **`x = 1`** to last line
  - But wait, p could be a pointer to x!

```
x = 1;
*p = 42;
y = 2 * pi * x;
```

- You should tell the compiler to what aliasing rules your code conforms
- If code does not conform to the rules: unexpected results

Aliasing in C

- ISO C defines rule for “strict aliasing”
  - Pointers of different type must not alias each other
- Compilers may rely on this rule at higher optimization levels
  - GNU: **`-O2`** enables **`-fstrict-aliasing`**
  - Intel: even **`-O3`** does not enable strict aliasing, do this with **`-ansi-alias`** or **`-fstrict-aliasing`**
- You can define even more strict aliasing rules
  - Function arguments do not alias each other, even if same type: **`-fargument-noalias`**
  - Additionally, arguments do not alias global storage: **`-fargument-noalias-global`**
  - Assume no aliasing at all (Intel only): **`-fno-alias`**, **`-fno-fnalias`**
- This allows more compiler optimizations but programmer must assure conformance to the rules!

Aliasing in C: `restrict` Keyword

- Keyword **`restrict`** is defined in C99
- A pointer declared as restrict must not be used to access other objects
  - Programmer is responsible to adhere to this rule
  - More compiler optimizations possible
- Requires the **`-std=c99`** compiler flag
- e.g. memcpy without overlapping memory areas:

```
void* memcpy(void restrict *dest, void restrict *src, size_t n)
```

Aliasing in Fortran

```
call func(x,x)
```

Illegal

- Less problematic than in C
- Subroutine arguments must not alias each other!
- More strict aliasing rules can be specified:
  - Assume no aliasing at all (Intel only): **`-fno-alias`**
  - Assume no aliasing within functions (Intel only): **`-fno-fnalias`**

Loop Unrolling

- Compiler can perform loop unrolling for you:
  - Intel: **`-unroll`**, enabled at **`-O2`**
  - GNU: **`-funroll-loops`**
- Only loops with known trip counts are unrolled (at compile time or upon entry to the loop)
  - Unroll all loops, may degrade performance: **`-funroll-all-loops`**
- More aggressive:
  - Intel: **`-unroll-aggressive`**

Floating Point Arithmetic

- Compiler optimizations
  - May affect accuracy of floating point arithmetic
  - May violate strict IEEE rules
- You can balance speed vs. accuracy using compiler flags
- Enable non-IEEE optimizations:
  - GNU: **`-ffast-math`**
  - Intel: **`-fp-model fast=1`** (default) or **`-fp-model fast=2`**
- Disable optimzations:
  - GNU: **`-fno-fast-math`** (default)
  - Intel: **`-fp-model precise -fp-model-source`** or **`-fp-model strict`**
- More options: read manual

Floating Point Arithmetic - Intel-only Flags

- All these flags violate IEEE semantics!
- Slightly less accurate but faster divisions: **`-no-prec-div`**
- Slightly less accurate but faster square roots: **`-no-prec-sqrt`**
- Slightly less accurate but faster sin, exp, ... **`-fast-transcendentals`**
- Flush SSE denormalized numbers (NaN, Inf) to zero: **`-ftz`**

Processor-specific Optimizations

- Tune code for Phobos CPUs (Opteron with SSE2)
  - GNU: **`-march=k8`**
  - Intel: **`-xW`**
  - This code will not run on a CPU without SSE2!
- Tune code for Deimos CPUs (Opteron with SSE3)
  - GNU: **`-march=k8 -msse3`**
  - Intel 11: **`-msse3`** / Intel <11: **`-xO`**
  - This code will not run on a CPU without SSE3!
- Tune code for CPU of compilation host
  - GNU: **`-march=native`** (only newer GNU compilers)
  - Intel 11: **`-xhost`**

Prefetching

- Prefetching = loading data from memory to CPU cache before the program actually needs it
- Goal: reduce processor stalls due to waiting for (slow) memory
- Useful when traversing large arrays
- Prefetching does not always improve performance
- Intel: **`-opt-prefetch`**
- GNU: **`-fprefetch-loop-arrays`**

Interprocedural Optimizations (IPO)

- Inlining, constant propagation, etc. across multiple files
- GNU: **-combine** (GCC 4.1 or higher)
  - Compiles all source files given in the command line at once, builds one combined object file
  - When building the whole program at once, use additionally **-fwhole-program** to allow further optimizations

```
gcc -O3 -combine -fwhole-program main.c utils.c -o myprog
```

- Intel: **-ipo**
  - Object files are compiled to intermediate representation
  - Avoid building libraries (or use Intel's **xiar**)

```
icc -c -O3 -ipo main.c

icc -c -O3 -ipo utils.c

icc -O3 -ipo main.o utils.o -o myprog
```

Actual compilation happens here

Profile-Guided Optimization (PGO)

- Based on an execution profile better optimizations are possible
  - 1: compile with profile generation
  - 2: run program with (small) representative data set
  - 3: compile again, use generated profiles
- Works best in combination with IPO
- GNU (GCC 3.4.6 or higher):
  - **`-fprofile-generate, -fprofile-use`**
- Intel:
  - **`-prof-gen, -prof-use`**
  - **`-prof-dir`** - specify directory where profiles are generated / read