How Compiling and Compilers Work

Dr. Axel Kohlmeyer

Research Professor, Department of Mathematics
Associate Director, Institute for Computational Science
Assistant Vice President for High-Performance Computing
Temple University, Philadelphia, USA

External Associate – HPC
International Centre for Theoretical Physics
Trieste, Italy
a.kohlmeyer@temple.edu

ICMS
Institute for Computational Molecular Science

Representing Real Numbers (1)

- Real numbers have unlimited accuracy
- Yet computers “think” digital, i.e. in integer math
  => only a fixed **range** of numbers can be
  represented by a fixed number of bits
  => **distance** between two integers is 1
- We can reduce the distance through fractions (= fixed point), but that also reduces the range

	16-bit	32-bit	64-bit	28-bit / 4-bit	22-bit / 10-bit
Min.	-32768	-2147483648	$\sim -9.2233 * 10^{-18}$	-16777216.0000	-2048.000000
Max.	32767	2147483647	$\sim 9.2233 * 10^{-18}$	16777215.9375	~ 2047.999023
Dist.	1	1	1	0.0635	0.0009765625

ICMS
Institute for Computational Molecular Science

Representing Real Numbers (2)

- We need a way to represent a wider range of numbers with a same number of bits
- We need a way to represent numbers with a reasonable amount of precision (distance)
- The same relative precision is often sufficient:

  => Scientific notation:

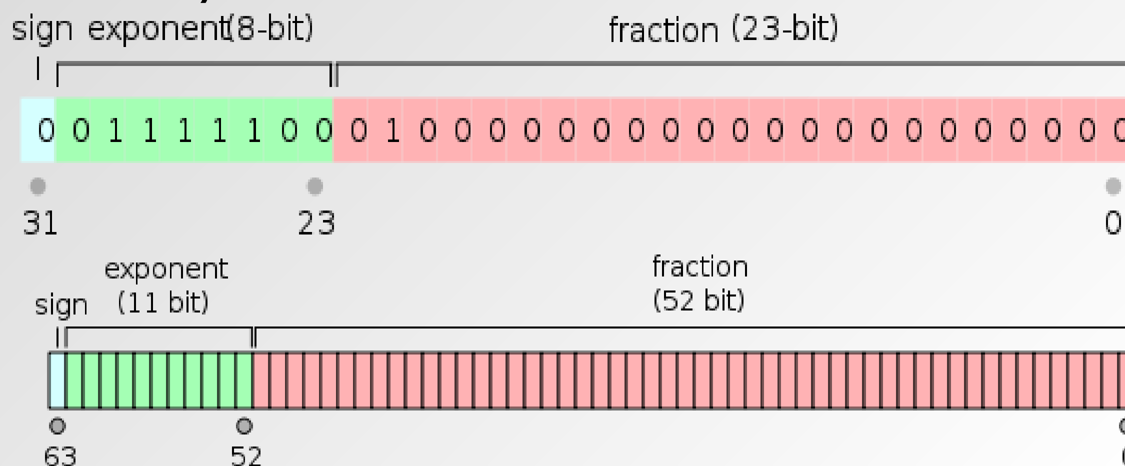
  $$\pm(\text{mantissa}) * (\text{base})^{\pm(\text{exponent})}$$

  Mantissa -> fixed point number (1.0 <= x < 2.0)
  Base -> 2
  Exponent -> a small integer (few bits)

ICMS
Institute for Computational Molecular Science

IEEE 754 Floating-point Numbers

- The IEEE 754 standard defines: storage format, result of operations, special values (infinity, overflow, invalid number), error handling => portability of compute kernels ensured
- Numbers are defined as bit patterns with a sign bit, an exponential field, and a fraction field
  - Single precision: 8-bit exponent 23-bit fraction
  - Double precision: 11-bit exponent 52-bit fraction



Values of Floating-Point Numbers

- Value: $(1 + \{mantissa\}/(2^{\{fraction\ bits\}}) * 2^{(\{exponent\}-\{bias\})}$
  $0 \leq \{mantissa\} < 2^{\{fraction\ bits\}}$, $\{exponent\} \geq 0$
- Special case: 0.0 is all bits set to zero
  Special case: -0.0 is like 0.0 but sign bit is set
  More special cases: Inf, -Inf, NaN, -NaN
- Single precision: $\sim\pm 1.2*10^{-38} < x < \sim\pm 3.4*10^{38}$
  relative precision: ~7 decimal digits
- Double precision: $\sim\pm 2.2*10^{-308} < x < \sim\pm 1.8*10^{308}$
  relative precision: ~15 decimal digits

ICMS Institute for Computational Molecular Science

Density of Floating-point Numbers

- How can we represent so many more numbers in floating point than in integer? ***<u>We don't!</u>***
- The number of unique bit patterns <u>has</u> to be the <u>same</u> as with integers of the same “bitness”
- There are 8,388,607 single precision numbers in 1.0< x < 2.0, but only 8191 in 1023.0< x <1024.0
- => absolute precision depends on the magnitude
- => some numbers are not represented exactly
=> approximated using rounding mode (nearest)

Floating-Point Math Pitfalls

- Floating point math is commutative,
  but **not** associative! Example (single precision):
  $1.0 + (1.5*10^{38} + (-1.5*10^{38})) = 1.0$
  $(1.0 + 1.5*10^{38}) + (-1.5*10^{38}) = 0.0$
- => the result of a summation depends on the order of how the numbers are summed up
- => results may change significantly, if a compiler changes the order of operations for optimization
- => prefer adding numbers of same magnitude
  => avoid subtracting very similar numbers

ICMS
Institute for Computational Molecular Science

Floating Point Comparison

- Floating-point results are usually **inexact**
  => comparing for equality is dangerous
  Example: don't use a floating point number for controlling a loop count. Integers are made for it
- It is OK to use exact comparison:
  - When results have to be bitwise identical
  - To prevent division by zero errors
- => compare against expected absolute error
- => don't expect higher accuracy than possible

Pre-process / Compile / Link

- Creating an executable includes multiple steps
- The “compiler” (gcc) is a wrapper for <u>several</u> commands that are executed in succession
- The “compiler flags” similarly fall into categories and are handed down to the respective tools
- The “wrapper” selects the compiler language from source file name, but links “its” runtime
- We will look into a C example first, since this is the language the OS is (mostly) written in

A simple C Example

- Consider the minimal C program 'hello.c':

```
#include <stdio.h>
int main(int argc, char **argv)
{
        printf(“hello world\n”);
        return 0;
}
```

- i.e.: what happens, if we do:

  > **gcc -o hello hello.c**

  (try: **gcc -v -o hello hello.c**)

ICMS Institute for Computational Molecular Science

Step 1: Pre-processing

- Pre-processing is <u>mandatory</u> in C (and C++)
- Pre-processing will handle '#' directives
  - File inclusion with support for nested inclusion
  - Conditional compilation and Macro expansion
- In this case: **`/usr/include/stdio.h`**
  - and all files are included by it - are inserted and the contained macros expanded
- Use -E flag to stop after pre-processing:
  > **`cc -E -o hello.pp.c hello.c`**

ICMS Institute for Computational Molecular Science

Step 2: Compilation

- Compiler converts a high-level language into the specific instruction set of the target CPU
- Individual steps:
  - Parse text (lexical + syntactical analysis)
  - Do language specific transformations
  - Translate to internal representation units (IRs)
  - Optimization (reorder,merge,eliminate,transform)
  - Replace IRs with chunks of assembly language
- Try:> **`gcc -S hello.c`** (produces **`hello.s`**)

ICMS
Institute for Computational Molecular Science

Compilation cont'd

```
        .file   "hello.c"
        .section    .rodata
.LC0:
        .string "hello, world!"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $16, %esp
        movl    $.LC0, (%esp)
        call    puts
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 4.5.1 20100924 (Red Hat 4.5.1-4)"
        .section        .note.GNU-stack,"",@progbits
```

gcc replaced `printf` with `puts`

try: gcc -fno-builtin -S hello.c

```
#include <stdio.h>
int main(int argc,
         char **argv)
{
 printf("hello world\n");
 return 0;
}
```

ICMS Institute for Computational Molecular Science

Step 3: Assembler / Step 4: Linker

- Assembler (as) translates assembly to binary
  - Creates so-called object files (in ELF format)

Try: > `gcc -c hello.c`
Try: > `nm hello.o`

```
00000000 T main
         U puts
```

- Linker (ld) puts binary together with startup code and required libraries
- Final step, result is executable.
  Try: > `gcc -o hello hello.o`

ICMS
Institute for Computational Molecular Science

Adding Libraries

- Example 2: exp.c

```
#include <math.h>
#include <stdio.h>
int main(int argc, char **argv)
{    double a=2.0;
     printf("exp(2.0)=%f\n", exp(a));
     return 0;
}
```

- `> gcc -o exp exp.c`
  Fails with “undefined reference to 'exp'”. Add: -lm
- `> gcc -O3 -o exp exp.c`
  Works due to inlining at high optimization level.

ICMS Institute for Computational Molecular Science

Symbols in Object Files & Visibility

- Compiled object files have multiple sections and a symbol table describing their entries:
  - “Text”: this is executable code
  - “Data”: pre-allocated variables storage
  - “Constants”: read-only data
  - “Undefined”: symbols that are used but not defined
  - “Debug”: debugger information (e.g. line numbers)
- Entries in the object files can be inspected with either the “nm” tool or the “readelf” command

ICMS Institute for Computational Molecular Science

Example File: visbility.c

```
static const int val1 = -5;
const int val2 = 10;
static int val3 = -20;
int val4 = -15;
extern int errno;

static int add_abs(const int v1, const int v2) {
    return abs(v1)+abs(v2);
}

int main(int argc, char **argv) {
     int val5 = 20;
     printf("%d / %d / %d\n",
            add_abs(val1,val2),
            add_abs(val3,val4),
            add_abs(val1,val5));
     return 0;
}
```

```
nm visibility.o:
00000000 t add_abs
         U errno
00000024 T main
         U printf
00000000 r val1
00000004 R val2
00000000 d val3
00000004 D val4
```

ICMS Institute for Computational Molecular Science

What Happens During Linking?

- Historically, the linker combines a “startup object” (crt1.o) with all compiled or listed object files, the C library (libc) and a “finish object” (crtn.o) into an executable (a.out)
- With current compilers it is more complicated
- The linker then “builds” the executable by matching undefined references with available entries in the symbol tables of the objects
- crt1.o has an undefined reference to “main” thus C programs start at the main() function

What is Different in Fortran?

- Basic compilation principles are the same
  => preprocess, compile, assemble, link
- In Fortran, symbols are <u>case insensitive</u>
  => most compilers translate them to lower case
- In Fortran symbol names may be modified to make them different from C symbols
  (e.g. append one or more underscores)
- Fortran entry point is not "main" (no arguments)
  PROGRAM => MAIN__ (in gfortran)
- C-like main() provided as startup (to store args)

ICMS
Institute for Computational Molecular Science

Pre-processing in C and Fortran

- Pre-processing is mandatory in C/C++
- Pre-processing is optional in Fortran
- Fortran pre-processing enabled implicitly via file name: name.F, name.F90, name.FOR
- Legacy Fortran packages often use /lib/cpp:
  /lib/cpp -C -P -traditional -o name.f name.F
  - -C : keep comments (may be legal Fortran code)
  - -P : no '#line' markers (not legal Fortran syntax)
  - -traditional : don't collapse whitespace (incompatible with fixed format sources)

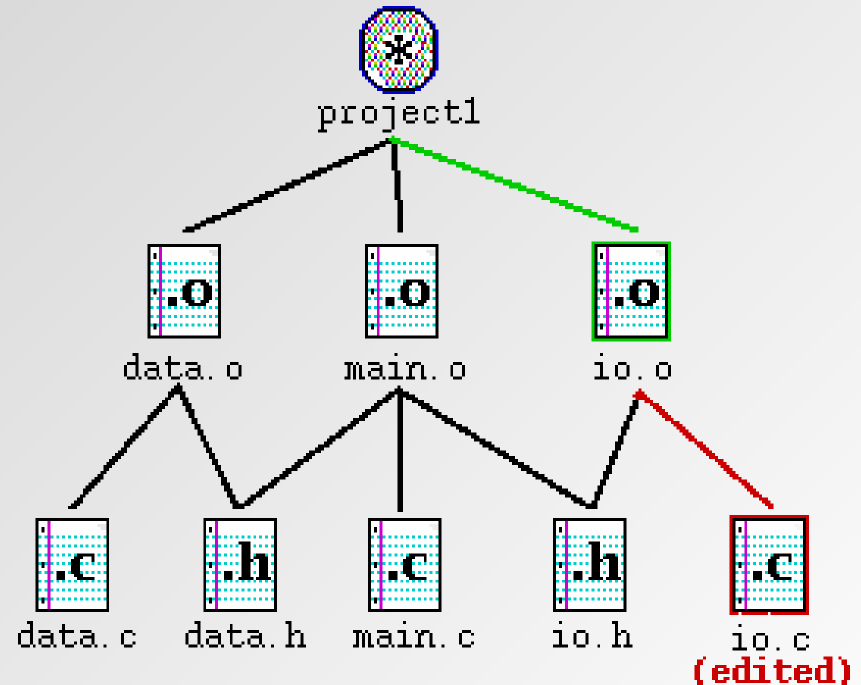
Common Compiler Flags

- Optimization: -O0, -O1, -O2, -O3, -O4, ...
  - Compiler will try to rearrange generated code so it executes faster
  - Aggressive compiler optimization may not always execute faster or may miscompile code
  - High optimization level (> 2) may alter semantics
- Preprocessor flags: -I/some/dir -DSOM_SYS
- Linker flags: -L/some/other/dir -lm
  -> search for libm.so/libm.a also in /some/dir

ICMS
Institute for Computational Molecular Science

Makefiles: Concepts

- Simplify building large code projects
- Speed up re-compile on small changes
- Consistent build command: make
- Platform specific configuration via Variable definitions



ICMS
Institute for Computational Molecular Science

Makefiles: Syntax

- Rules:

```
target: prerequisites
        command
```

^this must be a 'Tab' (|<- ->|)

- Variables:

```
NAME= VALUE1 VALUE2 value3
```

- Comments:

```
# this is a comment
```

- Special keywords:

```
include linux.mk
```

Makefiles: Rules Examples

```
# first target is default:
all: hello sqrt

hello: hello.c
       cc -o hello hello.c

sqrt: sqrt.o
       f77 -o sqrt sqrt.o
sqrt.o: sqrt.f
       f77 -o sqrt.o -c sqrt.f
```

ICMS Institute for Computational Molecular Science

Makefiles: Variables Examples

```
# uncomment as needed
CC= gcc
#CC= icc -i-static
LD=$(CC)
CFLAGS= -O2

hello: hello.o
       $(LD) -o hello hello.o

hello.o: hello.c
       $(CC)-c $(CFLAGS) hello.c
```

ICMS Institute for Computational Molecular Science

Makefiles: Automatic Variables

```
CC= gcc
CFLAGS= -O2

howdy: hello.o yall.o
       $(CC) -o $@ $^

hello.o: hello.c
       $(CC)-c $(CFLAGS) $<

yall.o: yall.c
       $(CC)-c $(CFLAGS) $<
```

Makefiles: Pattern Rules

```
OBJECTS=hello.o yall.o

howdy: $(OBJECTS)
       $(CC) -o $@ $^

hello.o: hello.c
yall.o: yall.c

.c.o:
       $(CC)-o $@ -c $(CFLAGS) $<
```

Rule to translate all XXX.c files to XXX.o files

ICMS Institute for Computational Molecular Science

Makefiles: Special Targets

```
.SUFFIXES:
.SUFFIXES: .o .F

.PHONY: clean install

.F.o:
	$(CPP) $(CPPFLAGS) $< -o $*.f
	$(FC)-o $@ -c $(FFLAGS) $*.f

clean:
	rm -f *.f *.o
```

Clear list of all known suffixes

Register new suffixes

Tell make to not look for theses files

ICMS
Institute for Computational Molecular Science

Makefiles: Calling make

- Override Variables:

```
make CC=icc CFLAGS='-O2 -unroll'
```

- Dry run (don't execute):

```
make -n
```

- Don't stop at errors (dangerous):

```
make -i
```

- Parallel make (requires careful design)

```
make -j2
```

- Use alternative Makefile

```
make -f make.pgi
```

ICMS
Institute for Computational Molecular Science