Performance Optimiziations for CPU Code

Dr. Axel Kohlmeyer

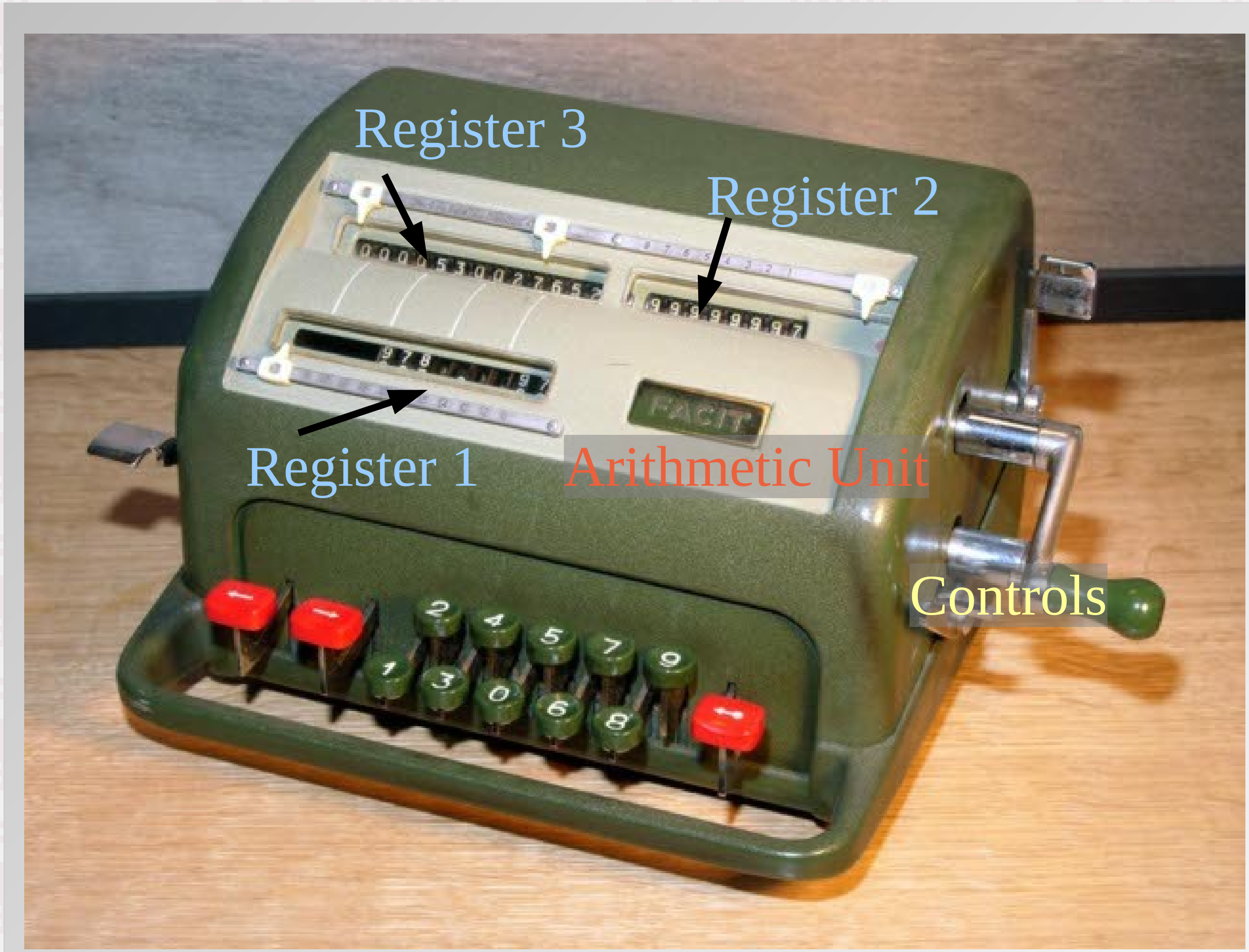
Associate Dean for Scientific Computing, CST
Associate Director, Institute for Computational Science
Assistant Vice President for High-Performance Computing

Temple University
Philadelphia PA, USA

a.kohlmeyer@temple.edu

ICMS
Institute for Computational Molecular Science

A Simple Calculator



1) Enter number on keyboard => register 1
2) Turn handle forward = add backward = subtract
3) Multiply = add register 1 with shifts until register 2 is 0
4) Register 3 = result

ICMS
Institute for Computational Molecular Science

Representing 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	~ $-9.2233 * 10^{-18}$	-16777216.0000	-2048.000000
Max.	32767	2147483647	~ $9.2233 * 10^{-18}$	16777215.9375	~ 2047.999023
Dist.	1	1	1	0.0635	0.0009765625

Institute for Computational Molecular Science

Representing Numbers (2)

- Need a way to represent a wider range of numbers with a same number of bits
- Need a way to represent numbers with a reasonable amount of precision (distance)
- Same relative precision often sufficient:

=> Scientific notation:

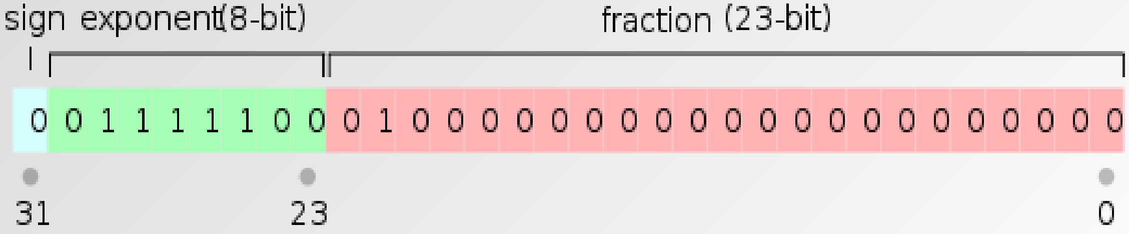
$$\pm(\text{mantissa}) * (\text{base})^{\pm(\text{exponent})}$$

Mantissa	-> integer fraction
Base	-> 2
Exponent	-> a small integer

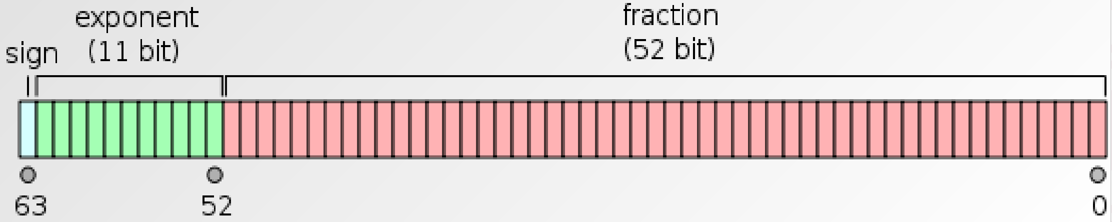
Institute for Computational Molecular Science

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



Density of Floating-point Numbers

- How can we represent so many more numbers in floating point than in integer? ***We don't!***
- The number of unique bit patterns has to be the same 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

How To Reduce Errors

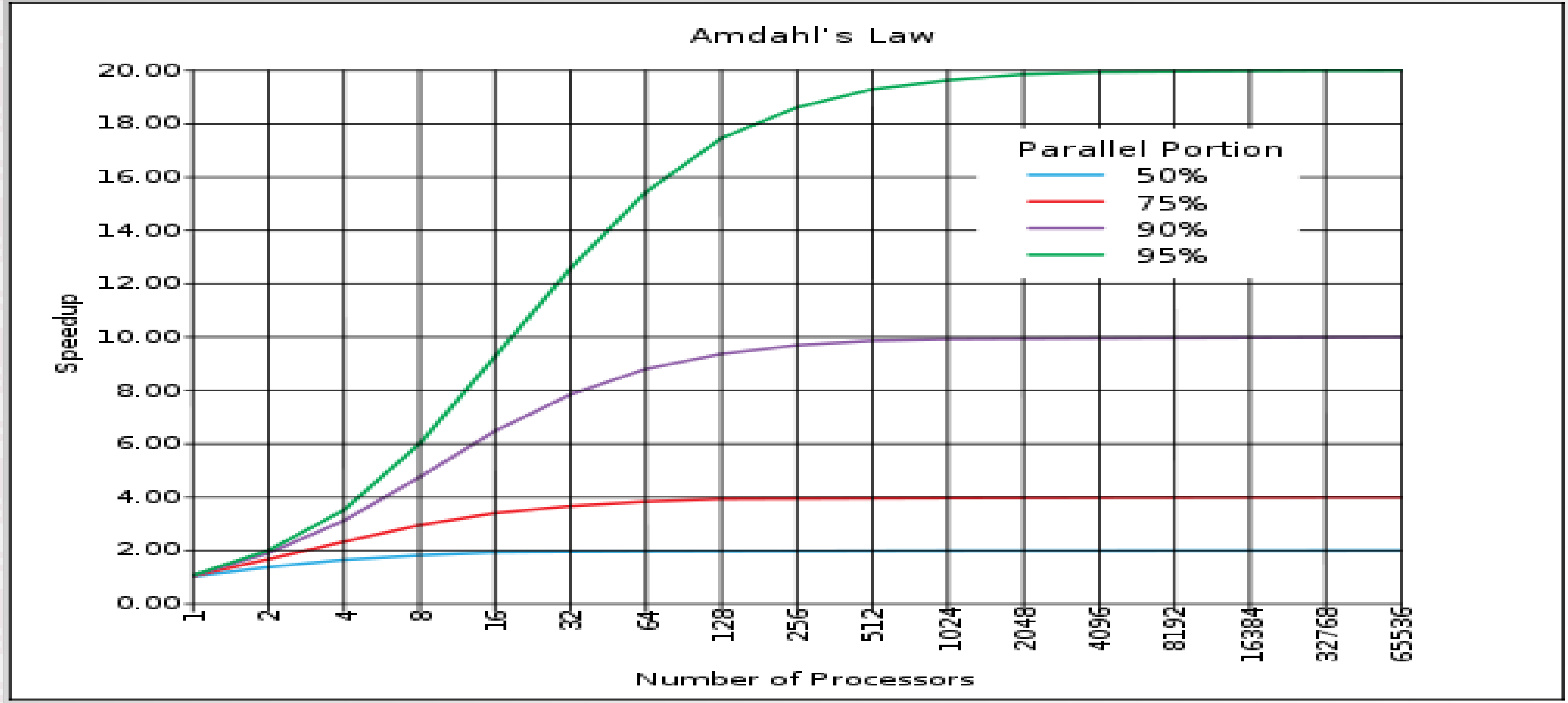
- Use double precision unless you can be sure of error cancellation or using an imprecise model => may collide with vectorization and GPU/MIC
- When summing numbers of different magnitude
  - Sort first and sum in ascending order
  - Sum in blocks (pairs) and then sum the sums
  - Use integer fraction, if range and precision allow it
- NOTE: summing numbers in parallel may give different results depending on parallelization

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

Reminder: Amdahl's Law

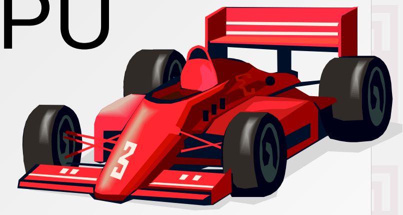
- The maximum speedup of a parallel code is limited by the fraction of sequential code.



Running Faster: Cache Memory

- Registers are very fast, but very **expensive**
- Loading data from memory is slow, but is is cheap and there can be a lot of it

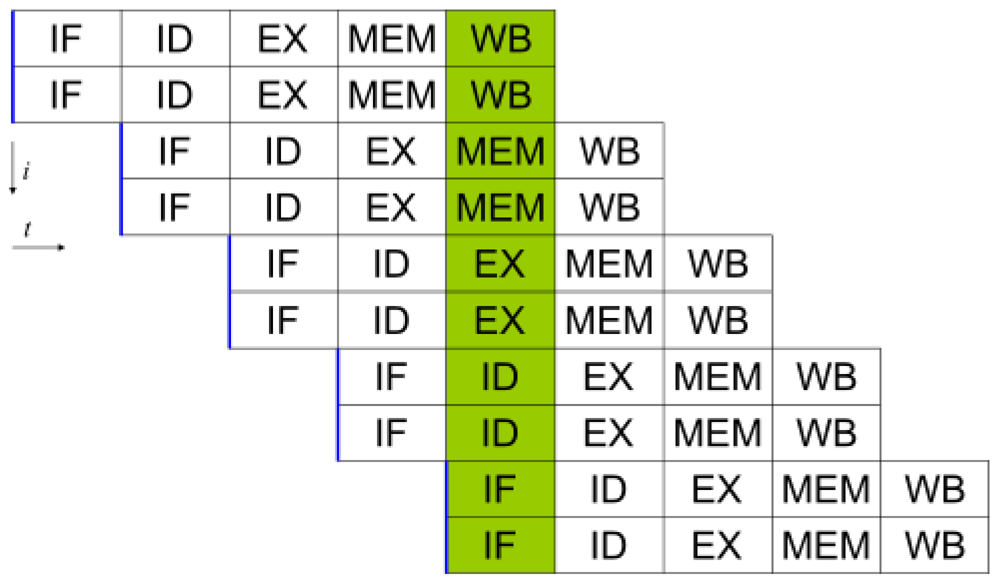
- => Cache memory = small buffer of fast memory that sits between RAM and CPU



- Cache memory is organized in “lines”:
  => when any byte is requested from RAM, a whole line (64 bytes) is read into the cache.
  => random memory access “pollutes” the cache

Running Faster: Superscalar CPU

- Superscalar CPU => instruction level parallelism
- Redundant functional units in single CPU
  => multiple instructions executed at same time,
  <u>**if**</u> there are no data dependencies
- Often combined with pipelined CPU design
- no branches
- Not SIMD/SSE/MMX
- Optimization:
  => loop unrolling



IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

Software Optimization

- Writing maximally efficient code is hard:
  => most of the time it will not be executed exactly as programmed, not even for assembly
- Maximally efficient code is not very portable:
  => cache sizes, pipeline depth, registers, instruction set will be different between CPUs
- Compilers are smart (but not too smart!) and can do the dirty work for us, but can get fooled

  => modular programming: generic code for most of the work plus well optimized kernels

How Would This Statement Be Executed on a Pipelined CPU?

z = a * b + c * d;

Actual steps:

z1 = a * b;

Data load can start while multiplying

z2 = c * d;

Start data load for next command

z= z1 + z2;

1. Load **a** into register **R0**
2. Load **b** into **R1**
3. Multiply **R2 = R0 * R1**
4. Load **c** into **R3**
5. Load **d** into **R4**
6. Multiply **R5 = R3 * R4**
7. Add **R6 = R2 + R5**
8. Store **R6** into **z**

Pipeline savings:
1 step out of 8, plus 3 more if next operation independent

Superscalar & Pipelined CPU Execution

```
z = a * b + c * d;
```

Actual steps:

z1 = a * b;

z2 = c * d;

Start data load for next command

z= z1 + z2;

1. Load **a** into register **R0** **and** load **b** into **R1**
2. Multiply **R2 = R0 * R1** **and** load **c** into **R3** **and** load **d** into **R4**
3. Multiply **R5 = R3 * R4**
4. Add **R6 = R2 + R5**
5. Store **R6** into **z**

Superscalar pipeline savings:
3 out of 8 steps, plus 3 if next operation independent

Superscalar & Pipelined Loop

```
for (i = 0; i < length; i++) {
  z[i] = a[i] * b[i] + c[i] * d[i];
}
```

1. Load `a[0]` into **R0** and load `b[0]` into **R1**
2. Multiply **R2 = R0 * R1** and load `c[0]` into **R3** and load `d[0]` into **R4**
3. Multiply **R5 = R3 * R4** and load `a[1]` into **R0** and load `b[1]` into **R1**
4. Add **R6 = R2 + R5** and load `c[1]` into **R3** and load `d[1]` into **R4**
5. Store **R6** into `z[0]` and multiply **R2 = R0 * R1** and multiply **R5 = R3 * R4** and load `a[2]` into **R0** and load `b[2]` into **R1**

Repeat steps 4. and 5. with increasing index until done
=> two steps per iteration

ICMS Institute for Computational Molecular Science

Vectorized Loop

```
for (i = 0; i < length; i++) {
  z[i] = a[i] * b[i] + c[i] * d[i];
}
```

Vector registers on a CPU can hold multiple numbers and load, store or process them in parallel (**SIMD**):

```
for (i = 0; i < length; i +=2) {
 z[i] = a[i]  *b[i]   + c[i]  *d[i];
 z[i+1]=a[i+1]*b[i+1] + c[i+1]*d[i+1];
}
```

Executed together

This is **<u>in addition</u>** to superscalar pipelining and with using special vector instructions (SSE,AVX,etc.)

Simple Optimization Techniques

(so easy a ~~caveman~~ compiler can do it)

ICMS Institute for Computational Molecular Science

Copy Propagation

Before

```
x = y
z = 1 + x
```

Has data dependency

Compile

After

```
x = y
z = 1 + y
```

No data dependency

Constant Folding

Before

```
add = 100;
aug = 200;
sum = add + aug;
```

After

```
sum = 300;
```

`sum` is the sum of two constants. The compiler can precalculate the result (once) at compile time and eliminate code that would otherwise need to be executed at (every) run time.

ICMS
Institute for Computational Molecular Science

Strength Reduction

Before

```
x = pow(y, 2);
a = c / 2.0;
```

After

```
x = y * y;
a = c * 0.5;
```

Raising one value to the power of another, or dividing, is more expensive than multiplying.

If the compiler can tell that the power is a small integer, or that the denominator is a constant, it will use multiplication instead.

Easier to do with intrinsic functions (cf. Fortran).

Common Subexpression Elimination

Before

```
d = c * (a / b);
e = (a / b) * 2.0;
```

After

```
adivb = a / b;
d = c * adivb;
e = adivb * 2.0;
```

The subexpression `(a / b)` occurs in both assignment statements, so there's no point in calculating it twice.

This is typically only worth doing if the common subexpression is expensive to calculate, or the resulting code requires the use of less registers.

Variable Renaming

Before

```
x = y * z;
q = r + x * 2;
x = a + b;
```

After

```
x0 = y * z;
q = r + x0 * 2;
x = a + b;
```

The original code has an **output dependency**, while the new code **doesn't** – but the final value of **`x`** is still correct.

Hoisting Loop Invariant Code

Code that doesn’t change inside the loop is known as ***loop invariant***. It doesn’t need to be calculated over and over.

Before

```
DO i = 1, n
  a(i) = b(i) + c * d
  e = g(n)
END DO
```

After

```
temp = c * d
DO i = 1, n
  a(i) = b(i) + temp
END DO
e = g(n)
```

Loop Unrolling

Before

```
DO i = 1, n
  a(i) = a(i)+b(i)
END DO
```

After

```
DO i = 1, n, 4
  a(i)   = a(i)  +b(i)
  a(i+1) = a(i+1)+b(i+1)
  a(i+2) = a(i+2)+b(i+2)
  a(i+3) = a(i+3)+b(i+3)
END DO
```

You generally **shouldn’t** unroll by hand.
Compilers are more reliable (no typos!).

Loop Interchange

Before

```
DO i = 1, ni
  DO j = 1, nj
    a(i,j) = b(i,j)
  END DO
END DO
```

After

```
DO j = 1, nj
  DO i = 1, ni
    a(i,j) = b(i,j)
  END DO
END DO
```

Array elements **`a(i,j)`** and **`a(i+1,j)`** are near each other in memory, while **`a(i,j+1)`** may be far, so it makes sense to make the **`i`** loop be the inner loop. (This is reversed in C, C++)

Inlining

Before

```
DO i = 1, n
  a(i) = func(i)
END DO
…
REAL FUNCTION func (x)
  …
  func = x * 3
END FUNCTION func
```

After

```
DO i = 1, n
  a(i) = i * 3
END DO
```

When a function or subroutine is ***inlined***, its contents are transferred directly into the calling routine, and thus eliminating the overhead of making the call.
=> compilers use an inline library at high optimization
=> math is instrinsic in Fortran => better for compiler

Pre-process / Compile / Link

- Creating an executable includes multiple steps
- The “compiler” (gcc) is a wrapper for several 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`)

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
  ```

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)
  - Replace IRs with pieces of assembler 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;
}
```

vector_add() Compilation

```
vector_add_cpu:
.LFB0:
        pushq   %rbp
        movq    %rsp, %rbp
        movq    %rdi, -40(%rbp)
        movq    %rsi, -48(%rbp)
        movq    %rdx, -56(%rbp)
        movl    %ecx, -60(%rbp)
        movq    -40(%rbp), %rax
        movq    %rax, -16(%rbp)
        movq    -48(%rbp), %rax
        movq    %rax, -24(%rbp)
        movq    -56(%rbp), %rax
        movq    %rax, -32(%rbp)
        movl    $0, -4(%rbp)
        jmp     .L2
.L3:
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -32(%rbp), %rax
        addq    %rax, %rdx
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rcx
        movq    -16(%rbp), %rax
        addq    %rcx, %rax
        movss   (%rax), %xmm1
        movl    -4(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rcx
        movq    -24(%rbp), %rax
        addq    %rcx, %rax
        movss   (%rax), %xmm0
        addss   %xmm0, %xmm1
        movd    %xmm1, %eax
        movl    %eax, (%rdx)
        addl    $1, -4(%rbp)
.L2:
        movl    -4(%rbp), %eax
        cmpl    -60(%rbp), %eax
        jl      .L3
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   vector_add_cpu, .-vector_add_cpu
        .ident  "GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
        .section        .note.GNU-stack,"",@progbits
```

```
void vector_add(float *a,
float *b,float *c,int dim)
{
  int i;
  for (i=0; i<dim; ++i)
    c[i] = a[i] + b[i];
}
```

ICMS
Institute for Computational Molecular Science

vector_add() w/ -O -mfpmath=387

```
        .file   "vector_add.c"
        .text
        .globl  vector_add_cpu
        .type   vector_add_cpu, @function
vector_add_cpu:
.LFB0:
        .cfi_startproc
        testl   %ecx, %ecx
        jle     .L1
        movl    $0, %eax
.L3:
        flds    (%rdi,%rax,4)
        fadds   (%rsi,%rax,4)
        fstps   (%rdx,%rax,4)
        addq    $1, %rax
        cmpl    %eax, %ecx
        jg      .L3
.L1:
        rep ret
        .cfi_endproc
.LFE0:
        .size   vector_add_cpu, .-vector_add_cpu
        .ident  "GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
        .section        .note.GNU-stack,"",@progbits
```

```
 void vector_add(float *a,
float *b,float *c,int dim)
{
  int i;
  for (i=0; i<dim; ++i)
    c[i] = a[i] + b[i];
}`
```

Same operations using the x86 floating point unit

vector_add() -O Compilation

```
        .file   "vector_add.c"
        .text
        .globl  vector_add_cpu
        .type   vector_add_cpu, @function
vector_add_cpu:
.LFB0:
        .cfi_startproc
        testl   %ecx, %ecx
        jle     .L1
        movl    $0, %eax
.L3:
        movss           (%rdi,%rax,4), %xmm0
        addss           (%rsi,%rax,4), %xmm0
        movss           %xmm0, (%rdx,%rax,4)
        addq    $1, %rax
        cmpl    %eax, %ecx
        jg      .L3
.L1:
        rep ret
        .cfi_endproc
.LFE0:
        .size   vector_add_cpu, .-vector_add_cpu
        .ident  "GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
        .section        .note.GNU-stack,"",@progbits
```

```
void vector_add(float *a,
float *b,float *c,int dim)
{
  int i;
  for (i=0; i<dim; ++i)
    c[i] = a[i] + b[i];
}`
```

Serial SSE instructions using SSE registers (exactly one)

vector_add() with SSE vectorization

```
vector_add_cpu:
.LFB0:
        testl   %ecx, %ecx
        jle     .L1
        leaq    16(%rdx), %r9
        cmpq%r9, %rdi
        setnb%r8b
    ( . . . )
        movl    $0, %eax
        movl    $0, %r9d
.L5:
        movaps      (%rdi,%rax), %xmm0
        addps       (%rsi,%rax), %xmm0
        movaps      %xmm0, (%rdx,%rax)
        addl    $1, %r9d
        addq $16, %rax
        cmpl    %r8d, %r9d
        jb      .L5
        jmp     .L15
.L7:
        movslq      %eax, %r8
        movss       (%rdi,%r8,4), %xmm0
        addss       (%rsi,%r8,4), %xmm0
        movss       %xmm0, (%rdx,%r8,4)
        addl    $1, %eax
        cmpl    %eax, %ecx
        jg      .L7
        rep ret
.
```

Parallel Instructions

```
L15:
        movl    %r10d, %eax
        cmpl    %r10d, %ecx
        jne     .L7
        rep ret
.L11:
        movl    $0, %eax
        jmp     .L7
.L10:
        movl    $0, %eax
.L3:
        movss       (%rdi,%rax,4), %xmm0
        addss       (%rsi,%rax,4), %xmm0
        movss       %xmm0, (%rdx,%rax,4)
        addq $1, %rax
        cmpl    %eax, %ecx
        jg      .L3
.L1:
        rep ret
        .cfi_endproc
.LFE0:
        .size   vector_add_cpu, .-vector_add_cpu
        .ident"GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
        .section        .note.GNU-stack,"",@progbits
```

ICMS
Institute for Computational Molecular Science

vector_add() with SSE vectorization

```
vector_add_cpu:
.LFB0:
        testl   %ecx, %ecx
        jle     .L1
        leaq    16(%rdx), %r9
        cmpq%r9, %rdi
        setnb%r8b
    ( . . . )
        movl  $0, %eax
        movl  $0, %r9d
.L5:
       vmovaps (%rdi,%r8), %ymm0
       vaddps  (%rsi,%r8), %ymm0, %ymm0
       vmovaps %ymm0, (%rdx,%r8)
       addl     $1, %r9d
       addq     $32, %r8
       cmpl     %eax, %r9d
       jb       .L5
       jmp      .L15
.L7:
       movslq  %eax, %r8
       vmovss  (%rdi,%r8,4), %xmm0
       vaddss  (%rsi,%r8,4), %xmm0, %xmm0
       vmovss  %xmm0, (%rdx,%r8,4)
       addl     $1, %eax
       cmpl     %eax, %ecx
       jg       .L7
       ret
```

Parallel Instructions

```
L15:
        movl  %r10d, %eax
        cmpl  %r10d, %ecx
        jne     .L7
        rep ret
.L11:
        movl  $0, %eax
        jmp    .L7
.L10:
        movl  $0, %eax
.L3:
       vmovss  (%rdi,%rax,4), %xmm0
       vaddss  (%rsi,%rax,4), %xmm0, %xmm0
       vmovss  %xmm0, (%rdx,%rax,4)
       addq     $1, %rax
       cmpl     %eax, %ecx
       jg       .L3
.L1:
        rep ret
        .cfi_endproc
.LFE0:
        .size  vector_add_cpu, .-vector_add_cpu
        .ident"GCC: (GNU) 4.9.2 20150212 (Red Hat 4.9.2-6)"
        .section      .note.GNU-stack,"",@progbits
```

Performance Comparison

- Running vector_add() for 10,000,000 elements:
  - No optimization: 35ms
  - Manual loop unrolling (4x): 25ms
  - Manual loop unrolling (8x): 24ms
  - Full optimization, gcc 4.9.x: 8.5ms
  - Full optimization + manual loop unrolling: 9.7ms
  - Full optimization, intel 13.x: 8.7ms

Matrix Multiply Optimization

- Need to access rows or matrix A and columns of matrix B multiple times => CPU cache
- Looping through columns of matrix B has strided access => cache pollution
- Lesson from GPU: use temporary buffer
- Change loop order and make loop over columns outer loop
- Copy column into auxiliary buffer
- Loop over rows and use buffer for dot product

Matrix Multiply Kernel Comparison

```
void matmul_cpu(float *a, float *b, float *c,
int n, int m, int o)
{
    int i,j,k;
    float sum;
    for (i = 0; i < n; ++i)
        for (j = 0; j < o; ++j) {
            sum = 0.0f;
            for (k = 0; k < m; ++k)
                sum += a[m*i+k] * b[o*k+j];

            c[o*i+j] = sum;
        }
}
```

Matrix Multiply Kernel Comparison

```
void matmul_opt(float *a, float *b, float *c,
 int n, int m, int o) {
    int i,j,k;
    float aux[m],sum;
    for (j = 0; j < o; ++j) {
        for (k = 0; k < m; ++k)
            aux[k] = b[o*k+j];
        for (i = 0; i < n; ++i) {
            sum = 0.0f;
            for (k = 0; k < m; ++k)
                sum += a[m*i+k] * aux[k];
            c[o*i+j] = sum;
        }
    }
}
```

Performance Comparison

- Running matrix_multiply() for 1000, 1024, 3000:
  - No compiler optimization: 18.8s
  - Same with buffer added: 10.7s
  - Same with OpenMP added: 5.2s (2 cores plus HT)
  - Full optimization, gcc 4.9.x: 9.6s
  - Same with buffer added: 3.5s
  - Same with OpenMP added: 1.3s
  - Full optimization, intel 13.x: 10.3s
  - Same with buffer added: 0.91s
  - Same with OpenMP added: 0.5s

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
```

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

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
```

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

Static Libraries

- Static libraries built with the “ar” command are collections of objects with a global symbol table
- When linking to a static library, object code is copied into the resulting executable and all direct addresses recomputed (e.g. for “jumps”)
- Symbols are resolved “from left to right”, so circular dependencies require to list libraries multiple times or use a special linker flag
- When linking only the name of the symbol is checked, not whether its argument list matches

Shared Libraries

- Shared libraries are more like executables that are missing the main() function
- When linking to a shared library, a marker is added to load the library by its “generic” name (soname) and the list of undefined symbols
- When resolving a symbol (function) from shared library all addresses have to be recomputed (relocated) on the fly.
- The shared linker program is executed first and then loads the executable and its dependencies

Differences When Linking

- Static libraries are fully resolved “left to right”; circular dependencies are only resolved between explicit objects or inside a library
-> need to specify libraries multiple times
or use: **-Wl,--start-group (...) -Wl,--end-group**
- Shared libraries symbols are **not** fully resolved at link time, only checked for symbols required by the object files. **Full check** only at runtime.
- Shared libraries may depend on other shared libraries whose symbols will be globally visible

Semi-static Linking

- Fully static linking is a bad idea with GNU libc; it <u>requires</u> matching shared objects for NSS
- Dynamic linkage of add-on libraries requires a compatible version to be installed (e.g. MKL)
- Static linkage of individual libs via linker flags

  ```
  -Wl,-Bstatic,-lfftw3,-Bdynamic
  ```
- can be combined with grouping, example:

  ```
  -Wl,--start-group,-Bstatic                    \
        -lmkl_gf_lp64 -lmkl_sequential    \
        -lmkl_core -Wl,--end-group,-Bdynamic
  ```

Dynamic Linker Properties

- Linux defaults to dynamic libraries:

  ```
  > ldd hello
  linux-gate.so.1 =>  (0x0049d000)
  libc.so.6 => /lib/libc.so.6
  (0x005a0000)
  /lib/ld-linux.so.2 (0x0057b000)
  ```

- **`/etc/ld.so.conf`, `LD_LIBRARY_PATH`** define where to search for shared libraries
- **`gcc -Wl,-rpath,/some/dir`** will encode **`/some/dir`** into the binary for searching

ICMS
Institute for Computational Molecular Science

Difference Between C and Fortran

- Basic compilation principles are the same
  => preprocess, compile, assemble, link
- In Fortran, symbols are case insensitive
  => 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)

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)

Fortran Symbols Example

```
 SUBROUTINE GREET
  PRINT*, 'HELLO, WORLD!'
END SUBROUTINE GREET

program hello
  call greet
end program
```

```
0000006d t MAIN__
         U _gfortran_set_args
         U _gfortran_set_options
         U _gfortran_st_write
         U _gfortran_st_write_done
         U _gfortran_transfer_character
00000000 T greet_
0000007a T main
```

- “program” becomes symbol “MAIN__” (compiler dependent)
- “subroutine” name becomes lower case with '_' appended
- several “undefineds” with '_gfortran' prefix
  => calls into the Fortran runtime library, libgfortran
- cannot link object with “gcc” alone, need to add -lgfortran
  => cannot mix and match Fortran objects from different compilers

ICMS
Institute for Computational Molecular Science

Fortran 90+ Modules

- When subroutines or variables are defined inside a module, they have to be hidden

```
module func
  integer :: val5, val6
contains
  integer function add_abs(v1,v2)
    integer, intent(in) :: v1, v2
    add_abs = iabs(v1)+iabs(v2)
  end function add_abs
end module func
```

- gfortran creates the following symbols:

```
00000000 T __func_MOD_add_abs
00000000 B __func_MOD_val5
00000004 B __func_MOD_val6
```

The Next Level: C++

- In C++ functions with different number or type of arguments can be defined (overloading) => encode prototype into symbol name:

  Example : symbol for `int add_abs(int,int)` becomes: `_ZL7add_absii`

- Note: the return type is not encoded
- C++ symbols are no longer compatible with C => add 'extern "C"' qualifier for C style symbols
- C++ symbol encoding is compiler specific

C++ Namespaces and Classes vs. Fortran 90 Modules

- Fortran 90 modules share functionality with classes and namespaces in C++
- C++ namespaces are encoded in symbols
  Example: `int func::add_abs(int,int)`
  becomes: `_ZN4funcL7add_absEii`
- C++ classes are encoded the same way
- Figuring out which symbol to encode into the object as undefined is the job of the compiler
- When using the gdb debugger use '::' syntax

Institute for Computational Molecular Science

Why We Need Header or Module Files

- The linker is “blind” for any language specific properties of a symbol => checking of the validity of the interface of a function is only possible during compilation
- A header or module file contains the prototype of the function (not the implementation) and the compiler can compare it to its use
- Important: header/module has to match library => Problem with FFTW-2.x: cannot tell if library was compiled for single or double precision

Calling C from Fortran 77

- Need to make C function look like Fortran 77
  - Append underscore (except on AIX, HP-UX)
  - Call by reference conventions
  - Best only used for “subroutine” constructs (cf. MPI) as passing return value of functions varies a lot:

    ```
    void add_abs_(int *v1,int *v2,int *res){
    *res = abs(*v1)+abs(*v2);}
    ```
- Arrays are always passed as “flat” 1d arrays by providing a pointer to the first array element
- Strings are tricky (no terminal 0, length added)

Calling C from Fortran 77 Example

```
void sum_abs_(int *in, int *num, int *out) {
 int i,sum;
 sum = 0;
 for (i=0; i < *num; ++i) { sum += abs(in[i]);}
   *out = sum;
   return;
}

/* fortran code:
   integer, parameter :: n=200
   integer :: s, data(n)

   call SUM_ABS(data, n, s)
   print*, s
*/
```

Calling Fortran 77 from C

- Inverse from previous, i.e. need to add underscore and use lower case (usually)
- Difficult for anything but Fortran 77 style calls since Fortran 90+ features need extra info
  - Shaped arrays, optional parameters, modules
- Arrays need to be “flat”,
  C-style multi-dimensional arrays are lists of pointers to individual pieces of storage, which may not be consecutive
  => use 1d and compute position

Calling Fortran 77 From C Example

```
subroutine sum_abs(in, num, out)
   integer, intent(in)  :: num, in(num)
   integer, intent(out) :: out
   Integer              :: i, sum
   sum = 0
   do i=1,num
     sum = sum + ABS(in(i))
   end do
   out = sum
end subroutine sum_abs
!! c code:
!   const int n=200;
!   int data[n], s;
!   sum_abs_(data, &n, &s);
!   printf("%d\n", s);
```

Modern Fortran vs C Interoperability

- Fortran 2003 introduces a standardized way to tell Fortran how C functions look like and how to make Fortran functions have a C-style ABI
- Module “iso_c_binding” provides kind definition: e.g. C_INT, C_FLOAT, C_SIGNED_CHAR
- Subroutines can be declared with “BIND(C)”
- Arguments can be given the property “VALUE” to indicate C-style call-by-value conventions
- String passing tricky, needs explicit 0-terminus

Calling C from Fortran 03 Example

```
int sum_abs(int *in, int num) {
  int i,sum;
  for (i=0,sum=0;i<num;++i) {sum += abs(in[i]);}
  return sum;
}
/* fortran code:
  use iso_c_binding, only: c_int
  interface
    integer(c_int) function sum_abs(in, num) bind(C)
      use iso_c_binding, only: c_int
      integer(c_int), intent(in) :: in(*)
      integer(c_int), value :: num
    end function sum_abs
  end interface
  integer(c_int), parameter :: n=200
  integer(c_int) :: data(n)
  print*, SUM_ABS(data,n)  */
```

Calling Fortran 03 From C Example

```
subroutine sum_abs(in, num, out) bind(c)
   use iso_c_binding, only : c_int
   integer(c_int), intent(in)  :: num,in(num)
   integer(c_int), intent(out) :: out
   integer(c_int),             :: i, sum
   sum = 0
   do i=1,num
     sum = sum + ABS(in(i))
   end do
   out = sum
end subroutine sum_abs

!! c code:
!   const int n=200;
!   int data[n], s;
!   sum_abs(data, &n, &s);
!   printf("%d\n", s);
```

Linking Multi-Language Binaries

- Inter-language calls via mutual C interface only due to name “mangling” of C++ / Fortran 90+
=> extern “C”, ISO_C_BINDING, C wrappers
- Fortran “main” requires Fortran compiler for link
- Global static C++ objects require C++ for link
=> avoid static objects (good idea in general)
- Either language requires its runtime for link
=> GNU: -lstdc++ and -lgfortran
=> Intel: “its complicated” (use -# to find out)
more may be needed (-lgomp, -lpthread, -lm)