Numeric Programming Examples

Thomas Hahn

Max-Planck-Institut für Physik
München

Topics

- Mixing Fortran and C
- MathLink Programming
- Floating-point issues
- Alignment and Caching
- "Find the Mistake" Quiz

Mixing Fortran and C

Why Fortran 77/90/2003? Why C/C++?

- Around for longer than many modern languages:
  Fortran 1957, C 1972
  Perl 1987, Python 1991, Java 1995, Ruby 1995
- Both widely used, e.g. C in the Linux Kernel.
- Good and free compilers available.
- Being the language of Unix, C is usually the lowest common denominator, i.e. has fewest linking issues.
- Object orientation through Fortran 90/2003, C++.
  (Introduces name mangling issues, though.)

Mixing Fortran and C

- Most Fortran compilers add an underscore to all symbols.
- Fortran passes all arguments by reference.
- Avoid calling functions (use subroutines) as handling of the return value is compiler dependent.
- 'Strings' are character arrays in Fortran and not null-terminated. For every character array the length is passed as an invisible int at the end of the argument list.
- Common blocks correspond to global structs, e.g.

```
double precision a, b
common /abc/ a, b
```

$\leftrightarrow$

```
struct {
  double a, b;
} abc_;
```

- Fortran's (and C99's) `double complex` maps onto `struct { double re, im; }`.

MathLink programming

MathLink is Mathematica's API to interface with C and C++.
J/Link offers similar functionality for Java.

A MathLink program consists of three parts:

a) Declaration Section

```
:Begin:
:Function: a0
:Pattern: A0[m_, opt___Rule]
:Arguments: {N[m], N[Delta /. {opt} /. Options[A0]],
   N[Mudim /. {opt} /. Options[A0]]}
:ArgumentTypes: {Real, Real, Real}
:ReturnType: Real
:End:

:Evaluate: Options[A0] = {Delta -> 0, Mudim -> 1}
```

MathLink programming

b) C code implementing the exported functions

```
#include "mathlink.h"

static double a0(const double m,
  const double delta, const double mudim)
{
  return m*(1 - log(m/mudim) + delta);
}
```

MathLink programming

c) Boilerplate main function

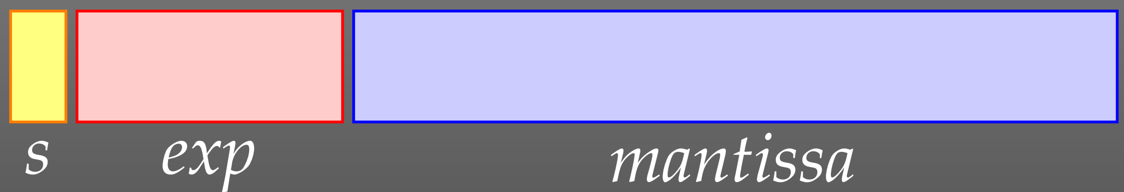
```
int main(int argc, char **argv)
{
  return MLMain(argc, argv);
}
```

Compile with `mcc` instead of `cc`.
Load in Mathematica with `Install["program"]`.

For even more details see arXiv:1107.4379.

Floating-point Representation

Floating-point numbers are these days always represented internally according to IEEE 754:



- s = sign bit,
- exp = (biased) exponent,
- $mantissa$ = (normalized) mantissa, i.e. implicit MSB = 1.

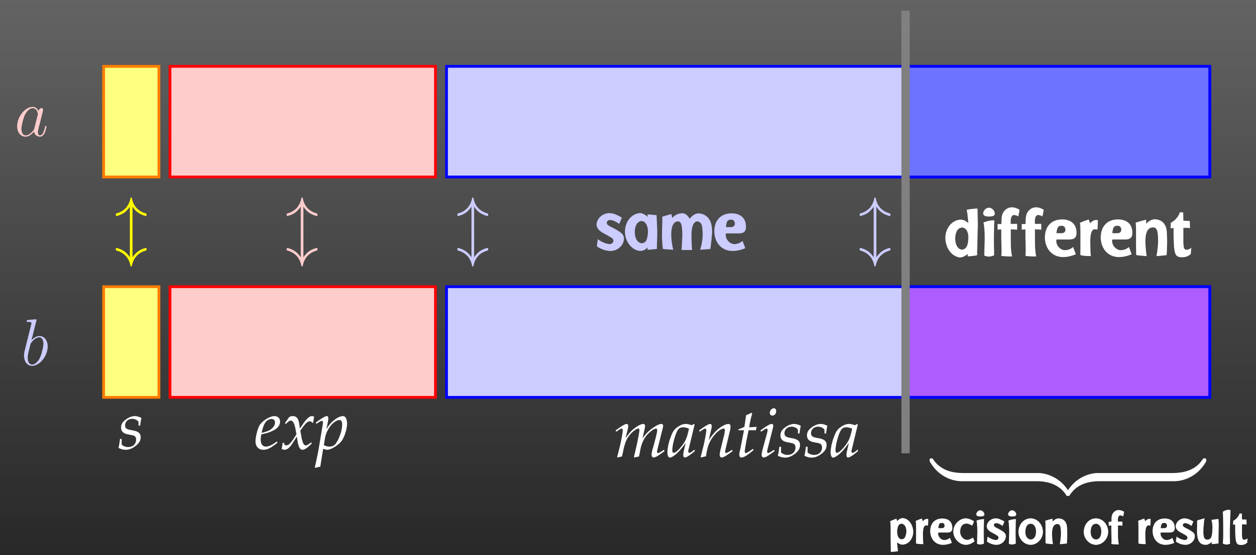
Special values	exponent	mantissa
Zero	0	0
Denormalized numbers	0	non-zero
Infinities	max	0
NaNs	max	non-zero

Bits	exponent	mantissa
Single precision	8	23
Double precision	11	52

Primitive Numerical Optimizations

About the only operation that can seriously cost precision in floating-point arithmetic is subtraction of two similar numbers,

$$a - b, \qquad |a - b| \ll |a| + |b|\,.$$



Primitive Numerical Optimizations

Numerical example for loss of precision:

$$\Delta p = p_0 - |\vec{p}\,| = \sqrt{p^2 + m^2} - p$$

p	m	$\Delta p^{\text{double precision}}$	Δp^{exact}
10^3	1	$.499999875046342 \cdot 10^{-3}$	$.499999875000062 \cdot 10^{-3}$
10^6	1	$.500003807246685 \cdot 10^{-6}$	$.499999999999875 \cdot 10^{-6}$
10^9	1	0	$.500000000000000 \cdot 10^{-9}$
10^{12}	1	0	$.500000000000000 \cdot 10^{-12}$
10^{15}	1	0	$.500000000000000 \cdot 10^{-15}$

Primitive Numerical Optimizations

Always substitute $a^2 - b^2 \to (a-b)(a+b)$.

$a^2 - b^2$ **loses twice as many digits** as $(a-b)(a+b)$!

Besides: $a^2 - b^2 = 2\,\text{mul}, 1\,\text{add}$, $(a-b)(a+b) = 1\,\text{mul}, 2\,\text{add}$.

Variants on this theme:

- **On-shell momentum** p:

$$p_0 - p = (p_0 - p)\,\frac{p_0 + p}{p_0 + p} = \frac{m^2}{p_0 + p}$$

- **Trigonometry in extreme forward/backward direction:**

$$1 - \cos x = (1 - \cos x)\,\frac{1 + \cos x}{1 + \cos x} = \frac{\sin^2 x}{1 + \cos x}$$

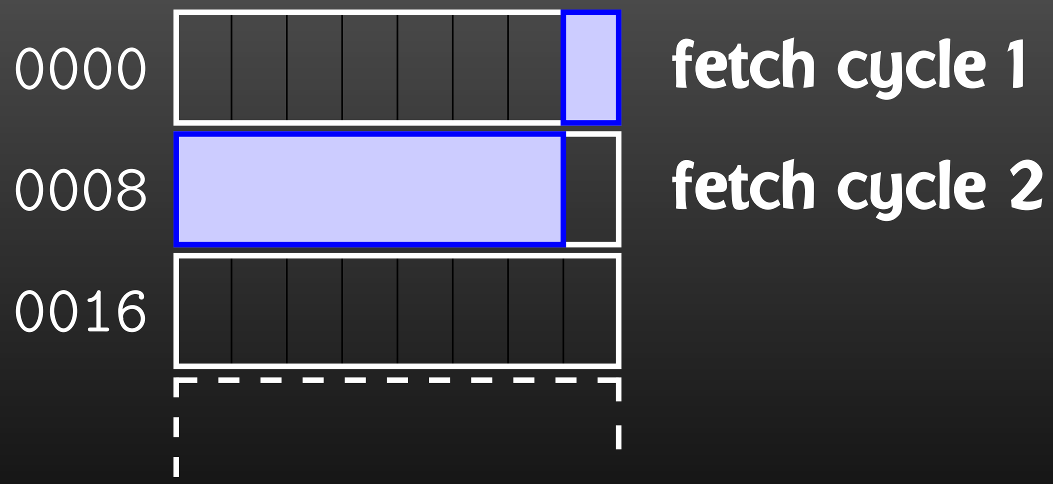
- **Polarization vectors:**

$$1 - e_z = (1 - e_z)\,\frac{1 + e_z}{1 + e_z} = \frac{e_x^2 + e_y^2}{1 + e_z}$$

Alignment

The CPU generally **accesses memory in units of its data bus width,** i.e. 4 bytes at a time on a 32-bit machine, 8 bytes at a times on a 64-bit machine.

If a variable is improperly aligned in memory, the CPU needs an **extra fetch cycle** to read the item! This significantly degrades performance.

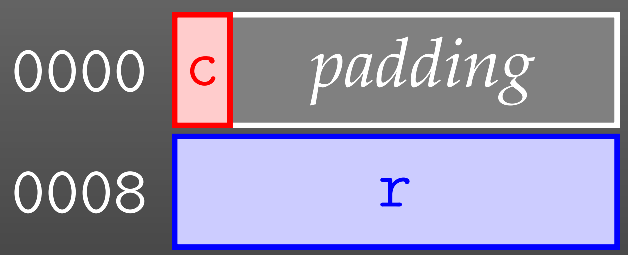


Alignment

Compilers generally align ‘loose’ variables on proper boundaries. Similarly, functions like `malloc` return memory addresses properly aligned for any type of data.

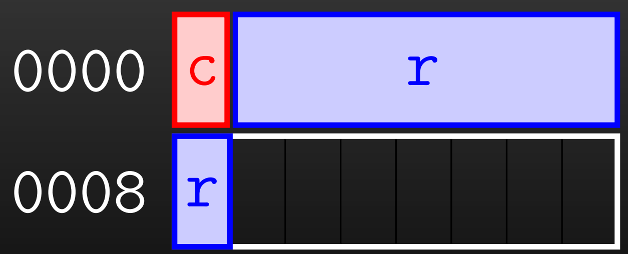
Some languages (e.g. C) **allow padding inside structures:**

```
struct test {
  char c;
  double r;  };
```



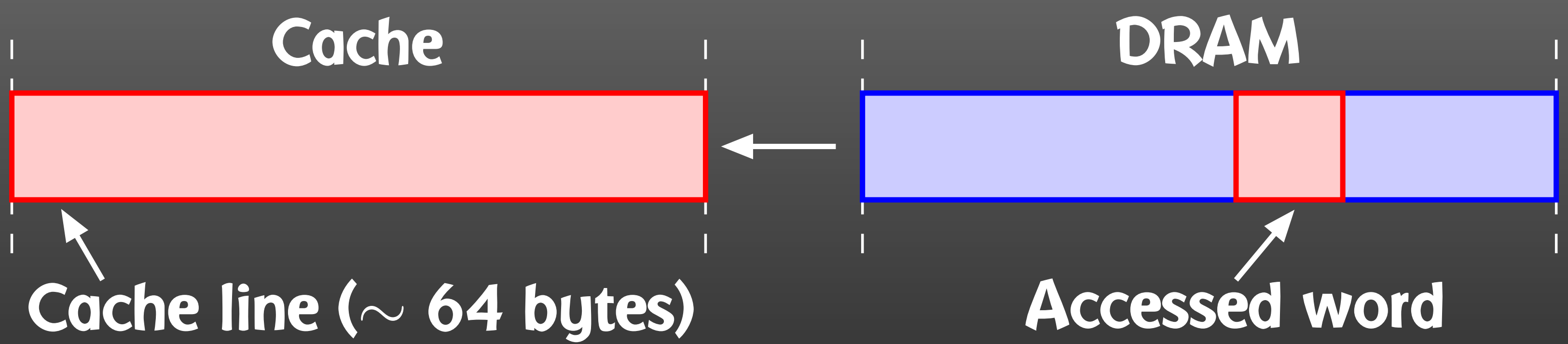
Some languages (e.g. Fortran) **do not allow padding,** thus the programmer can construct misaligned variables:

```
character*1 c
double precision r
common /test/ c, r
```



Cache

RAM (Random-Access Memory) is in fact not accessed randomly. Modern CPUs have **two levels of cache** “on top” of the regular RAM. Cache is much faster than DRAM, $t_{\text{cache}} : t_{\text{DRAM}} \approx t_{\text{DRAM}} : t_{\text{disk}}$.



Accessing memory sequentially is typically (much) faster than “hopping around.”

Matrix Storage

Due to the cache, accessing a matrix is not arbitrary.

- Fortran: column-major storage,
  Matrix = array of column vectors:
  $A_{11} \to A_{21} \to A_{31} \to \ldots$ (first index runs fastest)
- C: row-major storage,
  Matrix = array of row vectors:
  $A_{11} \to A_{12} \to A_{13} \to \ldots$ (last index runs fastest)

Naive:

```
do i = 1, n
  do j = 1, n
    sum = sum + A(i,j)
  enddo
enddo
```

Better:

```
do j = 1, n
  do i = 1, n
    sum = sum + A(i,j)
  enddo
enddo
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
inline double KineticEnergy(const double m,
                            const double v) {
  return 1/2*m*v*v;
}
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
program my_huge_program
double precision radius
print *, "Please enter the radius:"
read(*,*) radius
radius = radius*2*pi
...
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
subroutine foo(i)
integer i
i = 2*i + 1
end

...

call foo(4711)
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
#define map(a) 1-a
#define scale(x) 3*x+1

scaled_x = map(scale(x))
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
block data my_data_ini
double precision half, quarter
common /constants/ half, quarter
data half /1/2D0/
data quarter /1/4D0/
end
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
double precision x
character*1 id
double complex phase
common /mydata/ x, id, phase
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
subroutine foo(x)
double precision x
print *, x
end

...

call foo(7.2)
```

Find the Mistake

Can you spot what is wrong, undesirable, or potentially dangerous with the following code snippet?

```
program compute_sum
double precision x(5), sum
integer i
data x /1D40, 4.71D0, -2.5D40, 200D-2, 1.5D40/

sum = 0
do i = 1, 5
  sum = sum + x(i)
enddo
print *, sum
end
```