



## **Vincenzo Innocente:**

### **“Optimal floating point computation”**

#### ***Accuracy, Precision, Speed in scientific computing***

- IEEE 754 standard
- Expression optimization
- Approximate Math
  
- X86\_64 SIMD instructions
- Vectorization using the GCC compiler

# Objectives

- Understand precision and accuracy in floating point calculations
- Manage optimization, trading precision for speed (or vice-versa!)
- Understand and Exploit vectorization
- Learn how to make the compiler to work for us

---

# Prepare for the exercises

```
git clone https://github.com/VinInn/FPOptimization.git
```

```
Source gcc49env
```

```
which c++; c++ -v
```

```
cd FPOptimization
```

```
source compile exercises/afloat.cpp
```

```
./afloat
```

```
ldd ./afloat | grep libstdc
```

```
libstdc++.so.6 => /opt/rh/devtoolset-3/root/usr/lib64/libstdc++.so.6
```

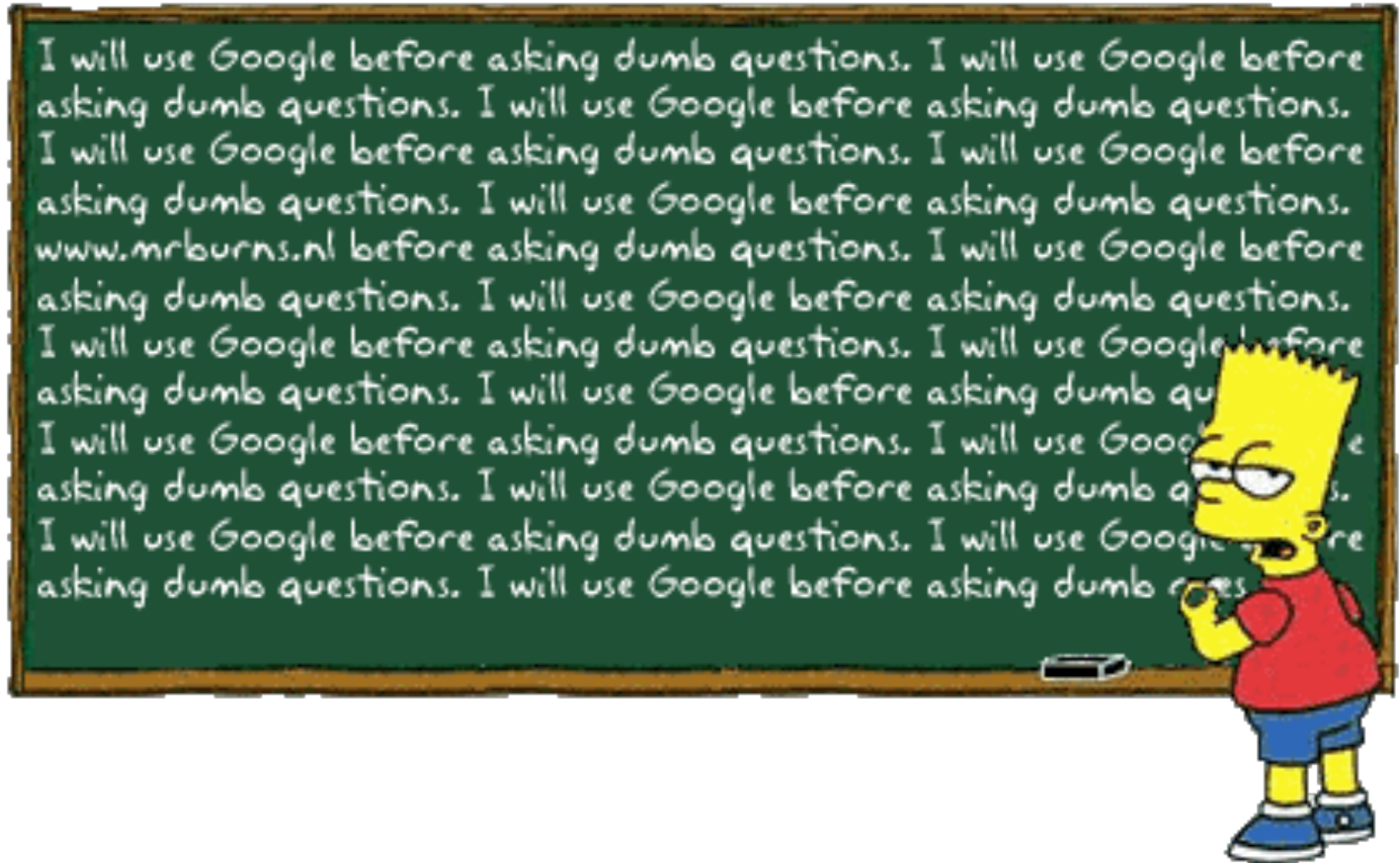
# Disclaimer, caveats, references

- This is NOT a full overview of IEEE 754
- It applies to x86\_64 systems and gcc > 4.7.0
  - Other compilers have similar behavior, details differ
- General References
  - Wikipedia has excellent documentation on the IEEE 754, math algorithms and their computational implementations
  - Handbook of Floating-Point Arithmetic (as google book)
  - Kahan home page
  - Jeff Arnold's (et al) seminar and course at CERN
  - INTEL doc and white papers
  - Ulrich Drepper recent talks

# Applicability

- It is very (very) different if you deal with
  - Video games
  - offline analysis of scientific data
  - Financial applications (with legal bindings)
  - Real time applications
  - Human-life
- <http://www.heidelberg-laureate-forum.org/blog/video/lecture-thursday-september-26-william-morton-ka-han/>

# Don't be afraid to ask questions!



# Floating behaviour...

```
void i() {  
    unsigned int w=0, y=0;  
    do { y = w++; } while (w>y);  
    std::cout << y << " " << w << std::endl;  
}
```

What's going on?  
What's the difference?  
Can you guess the result?

```
void f() {  
    int n=0; float w=1; float y=0; float prev=0;  
    do {  
        prev=y; y = w++;  
        ++n;  
    } while (w>y);  
    std::cout << n << ": " << prev << " " << y << " " << w << std::endl;  
}
```

```
c++ -O2 intLoop.cpp; ./a.out  
c++ -O2 floatLoop.cpp; ./a.out
```

```
4294967295 0  
16777216: 1.67772e+07 1.67772e+07 1.67772e+07  
0x1.fffffe+23 0x1p+24 0x1p+24
```

# Floating Point Representation

(source Wikipedia)

- floating point describes a system for representing numbers that would be too large or too small to be represented as integers.
  - The advantage of floating-point representation over fixed-point (and integer) representation is that it can support a much wider range of values.
    - int:  $-2,147,483,648$  to  $+2,147,483,647$ ,  $-(2^{31}) \sim (2^{31}-1)$
    - float:  $1.4 \times 10^{-45}$  to  $3.4 \times 10^{38}$
  - This has a cost...



# Floating behaviour...

```
include<cstdio>
int main() {
    float tenth=0.1f;
    float t=0;
    long long n=0;
    while(n<1000000) {
        t+=0.1f; // nanosleep omitted...
        ++n;
        if (n<21 || n%36000==0) printf("%d %f %a\n",n,t,t);
    }
    return 0;
}
```

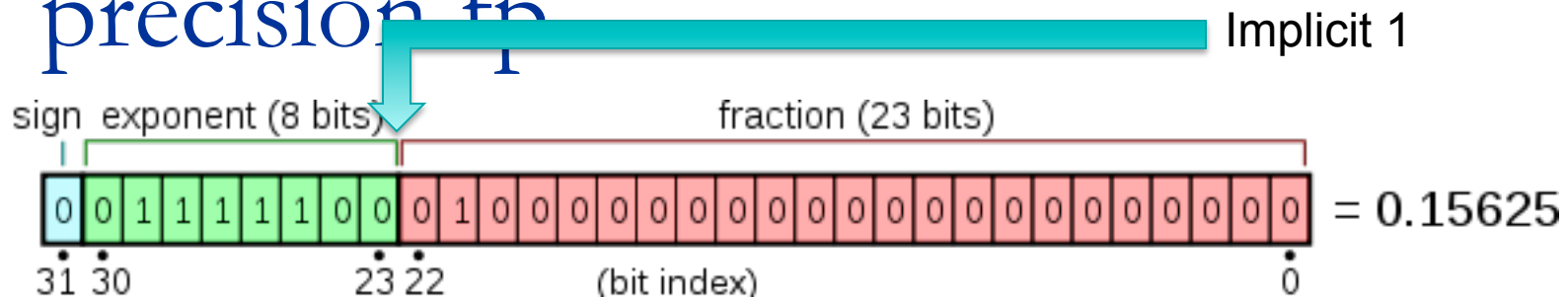
Not a swiss clock...  
Why?

```
cat patriot.cpp
c++ -std=c++1y -O2\
patriot.cpp
./a.out
```

# Patriot result

1	0.100000	0x1.99999ap-4
2	0.200000	0x1.99999ap-3
5	0.500000	0x1p-1
10	1.000000	0x1.000002p+0
20	2.000000	0x1.000002p+1
36000	3601.162354	0x1.c22532p+11
72000	7204.677734	0x1.c24ad8p+12
972000	98114.593750	0x1.7f4298p+16

# IEEE 754 representation of single precision fp



$$n = (-1)^s \times (m2^{-23}) \times 2^{x-127}$$

Exponent	significand zero	significand non-zero	Equation
00 <sub>H</sub>	zero, -0	subnormal numbers	$(-1)^{\text{signbits}} \times 2^{-126} \times 0.\text{significandbits}$
01 <sub>H</sub> , ..., FE <sub>H</sub>	normalized value	normalized value	$(-1)^{\text{signbits}} \times 2^{\text{exponentbits}-127} \times 1.\text{significandbits}$
FF <sub>H</sub>	±infinity	<u>NaN (quiet,signalling)</u>	

# Look into a float

```
void look(float x) {  
    printf("%a\n",x);  
    int e; float r = ::frexpf(x,&e); //  $0.5 < r < 1.0$  (or 0)  
    std::cout << x << " exp " << e << " res " << r << std::endl;  
  
    auto ftoi = [](float f)->int { int i; memcpy(&i,&f,4); return i;};  
  
    auto bin = ftoi(x);  
    printf("%e %a %x\n", x, x, bin);  
    auto log_2 = ((bin >> 23) & 255) - 127; //exponent  
    bin &= 0x7FFFFFFF; //mantissa (aka significand)  
  
    std::cout << "exp " << log_2 << " significand in binary " << std::hex << bin  
        << " significand as float " << std::dec << (bin|0x800000)*::pow(2.,-23)  
        << std::endl; // or scalbn((bin|0x800000),-23);  
}
```

Use printf.cpp

exercise: count the number of “floats” between two of them...

# Limits

- Everything you want to know about an arithmetic type is in
  - ❑ `std::numeric_limits<T>`
  - ❑ see [http://www.cplusplus.com/reference/limits/numeric\\_limits/](http://www.cplusplus.com/reference/limits/numeric_limits/)

```
#include <iostream>    // std::cout
#include <limits>       // std::numeric_limits

int main () {
    std::cout << std::boolalpha;
    std::cout << "Minimum value: " << std::numeric_limits<float>::min() << '\n';
    std::cout << "Maximum value: " << std::numeric_limits<float>::max() << '\n';
    std::cout << "Is signed: " << std::numeric_limits<float>::is_signed << '\n';
    std::cout << "Non-sign bits: " << std::numeric_limits<float>::digits << '\n';
    std::cout << "has infinity: " << std::numeric_limits<float>::has_infinity << '\n';
    return 0;
}
```

# Precision

```
float a = 100.f+3.f/7.f;  
float b = 4.f/7.f; // (in general |a|>|b|)
```

```
float s = a+b;  
float z = s-a;  
float t = b-z;
```

```
float w=s;  
for (auto i=0; i<1000000; ++i) w+=t;  
w=0;  
for (auto i=0; i<1000000; ++i) w+=t;
```

**Sterbenz Lemma:**  
If  $y/2 \leq x \leq 2*y$  then  $x-y$  is exact

What's the value of  $t$ ?  
How big is “ $t$ ” (w.r.t. “ $s$ ”)?

```
cat precision.cpp  
c++ -O2 precision.cpp  
./a.out
```

```
a=100.429 b=0.571429 s=101 z=0.571426 t=2.20537e-06  
a=0x1.91b6dcp+6 b=0x1.24924ap-1  
s=0x1.94p+6 z=0x1.2492p-1 t=0x1.28p-19  
101 = 0x1.94p+6  
nextafterf(s,maxf) = 101 0x1.940002p+6  
nextafterf(s,maxf)-s = 7.62939e-06 0x1p-17  
w= 101 0x1.94p+6  
w= 2.17279 0x1.161e2p+1
```

# Extending precision (source **Handbook of Floating-Point Arithmetic** pag 126)

```
void fast2Sum(T a, T b, T& s, T& t) {  
    if (std::abs(b) > std::abs(a)) std::swap(a,b);  
    // Don't allow value-unsafe optimizations  
    s = a + b;  
    T z = s - a;  
    t = b - z;  
    return;  
}
```

- $s+t = a+b$  exactly

- (s=a+b rounded to half ulp, t is the part of (a+b) in such half ulp)

# Kahan summation algorithm (source [http://en.wikipedia.org/wiki/Kahan\\_summation\\_algorithm](http://en.wikipedia.org/wiki/Kahan_summation_algorithm))

```
T kahanSum(T const * input, size_t n)
T sum = input[0];
T t = 0.0;      // A running compensation for lost low-order bits.
for (size_t i = 1; i!=n; ++i) {
    y = input[i] - t;      // so far, so good: t is zero.
    s = sum + y;           // Alas, sum is big, y small, so low-order digits of y are lost.
    t = (s - sum) - y;     // ( s - sum) recovers the high-order part of y
                          // subtracting y recovers -(low part of y)
    sum = s;               //Algebraically, t should always be zero.
                          // Beware eagerly optimising compilers!
}                          //Next time around, the lost low part will be added to y in a fresh attempt.
return sum;
```

Exercise: Apply this algorithm to the patriot.cpp



# Dekker Multiplication (source Handbook of Floating-Point Arithmetic pag 135)

```
template<typename T, int SP> inline void vSplit(T x, T & x_high, T & x_low) __attribute__((always_inline));
template<typename T, int SP> inline void vSplit(T x, T & x_high, T & x_low) {
    const unsigned int C = ( 1 << SP ) + 1;
    T a = C * x;
    T b = x - a;
    x_high = a + b;  x_low = x - x_high; // x+y = x_high + x_low exactly
}
template <typename T> struct SHIFT_POW{};
template <> struct SHIFT_POW<float>{ enum {value=12}; /* 24/2 for single precision */ };
template <> struct SHIFT_POW<double>{ enum {value = 27}; /* 53/2 for double precision */ };

template<typename T> inline void dMultiply(T x, T y, T & r1, T & r2) __attribute__((always_inline));
template<typename T> inline void dMultiply(T x, T y, T & r1, T & r2) {
    T x_high, x_low, y_high, y_low;
    vSplit<T,SHIFT_POW<T>::value>(x, x_high, x_low);
    vSplit<T,SHIFT_POW<T>::value>(y, y_high, y_low);
    r1 = x * y; // rounded
    T a = -r1 + x_high * y_high;
    T b =  a + x_high * y_low;
    T c =  b + x_low  * y_high;
    r2 = c + x_low  * y_low; // x*y = r1 + r2 exactly
}
```

# Floating Point Math

- Floating point numbers are NOT real numbers
  - They exist in a finite number ( $\sim 2^{32}$  for single prec)
    - Exercise: count how many floats exists between given two
  - Exist a “next” and a “previous” (`std::nextafter`)
    - Differ of one ULP (Unit in the Last Place or Unit of Least Precision [http://en.wikipedia.org/wiki/Unit\\_in\\_the\\_last\\_place](http://en.wikipedia.org/wiki/Unit_in_the_last_place))
  - Results of Operations are rounded
    - Standard conformance requires half-ULP precision.
    - $x + \varepsilon - x \neq \varepsilon$  (can easily be 0 or  $\infty$ )
  - Their algebra is not associative
    - $(a+b) + c \neq a+(b+c)$
    - $a/b \neq a*(1/b)$
    - $(a+b)*(a-b) \neq a^2 - b^2$

# Strict IEEE754 vs “Finite” (fast) Math

- Compilers can treat FP math either in “strict IEEE754 mode” or optimize operations using “algebra rules for finite real numbers” (as in FORTRAN)
  - gcc <= 4.5 `-funsafe-math -ffast-math`
  - gcc >= 4.6 `-Ofast` (switch vectorization on as well)
    - Caveat: the compiler improves continuously: still it does not optimize yet all kind of expressions
- <https://gcc.gnu.org/wiki/FloatingPointMath>

# Inspecting generated code

`objdump -S -r -C --no-show-raw-insn -w kernel.o | less`

(on MacOS: `otool -t -v -V -X kernel.o | c++filt | less`)

Or use <http://gcc.godbolt.org>

`c++ -S kernel.cc; less kernel.s`

**kernel(float, float, float):**

```
mulss    %xmm0,%xmm2
mulss    %xmm0,%xmm1
addss    %xmm2,%xmm1
movaps    %xmm1,%xmm0
ret
```

```
float
kernel(float a, float x, float y)
{
    return a*x + a*y;
}
```

-O2

-Ofast

**kernel(float, float, float):**

```
addss    %xmm2,%xmm1
mulss    %xmm0,%xmm1
movaps    %xmm1,%xmm0
ret
```

Exercise: `assocMath.cc`; compare assembler for O2 and Ofast

# Rounding Algorithms

- The standard defines five rounding algorithms.
  - The first two round to a nearest value; the others are called directed roundings:
- Roundings to nearest
  - **Round to nearest, ties to even – rounds to the nearest value;** if the number falls midway it is rounded to the nearest value with an even (zero) least significant bit, which occurs 50% of the time; **this is the default algorithm for binary floating-point** and the recommended default for decimal
  - Round to nearest, ties away from zero – rounds to the nearest value; if the number falls midway it is rounded to the nearest value above (for positive numbers) or below (for negative numbers)
- Directed roundings
  - Round toward 0 – directed rounding towards zero (also known as truncation).
  - Round toward  $+\infty$  – directed rounding towards positive infinity (also known as rounding up or ceiling).
  - Round toward  $-\infty$  – directed rounding towards negative infinity (also known as rounding down or floor).

# Floating point exceptions

The IEEE floating point standard defines several exceptions that occur when the result of a floating point operation is unclear or undesirable. Exceptions can be ignored, in which case some default action is taken, such as returning a special value. When trapping is enabled for an exception, an error is signalled whenever that exception occurs. These are the possible floating point exceptions:

- ❑ **Underflow:** This exception occurs when the result of an operation is too small to be represented as a normalized float in its format. If trapping is enabled, the *floating-point-underflow* condition is signalled. Otherwise, the operation results in a denormalized float or zero.
- ❑ **Overflow:** This exception occurs when the result of an operation is too large to be represented as a float in its format. If trapping is enabled, the *floating-point-overflow* exception is signalled. Otherwise, the operation results in the appropriate infinity.
- ❑ **Divide-by-zero:** This exception occurs when a float is divided by zero. If trapping is enabled, the *divide-by-zero* condition is signalled. Otherwise, the appropriate infinity is returned.
- ❑ **Invalid:** This exception occurs when the result of an operation is ill-defined, such as  $(0.0/0.0)$ . If trapping is enabled, the *floating-point-invalid* condition is signalled. Otherwise, a quiet NaN is returned.
- ❑ **Inexact:** This exception occurs when the result of a floating point operation is not exact, i.e. the result was rounded. If trapping is enabled, the *floating-point-inexact* condition is signalled. Otherwise, the rounded result is returned.

# Gradual underflow (subnormals)

- *Subnormals* (or *denormals*) are fp smaller than the smallest normalized fp: they have leading zeros in the significand
  - For single precision they represent the range  $10^{-38}$  to  $10^{-45}$
- Subnormals guarantee that additions never underflow
  - Any other operation producing a *subnormal* will raise a underflow exception if also inexact
- Literature is full of very good reasons why “gradual underflow” improves accuracy
  - This is why they are part of the IEEE 754 standard
- Hardware is not always able to deal with *subnormals*
  - Software assist is required: **SLOW**
  - To get correct results even the software algorithms need to be specialized
- It is possible to tell the hardware to *flush-to-zero* subnormals
  - It will raise underflow and inexact exceptions

# Cost of operations (in cpu cycles)

op	instruction	sse s	sse d	avx s	avx d
+, -	ADD, SUB	3	3	3	3
== < >	COMISS CMP..	2,3	2,3	2,3	2,3
f=d d=f	CVT..	3	3	4	4
, &, ^	AND, OR	1	1	1	1
*	MUL	5	5	5	5
/, sqrt	<b>DIV, SQRT</b>	<b>10-14</b>	<b>10-22</b>	<b>21-29</b>	<b>21-45</b>
1.f/ , 1.f/sqrt	RCP, RSQRT	5		7	
=	MOV	1,3,...	1,3,...	1,4,....	1,4,....

→ 350 from  
main memory



# Approximate reciprocal (sqrt, div)

The major contribution of game industry to SE is the discovery of the “magic” fast  $1/\sqrt{x}$  algorithm

```
float invSqrt(float x){  
    int i; memcpy(&i,&x,4);  
    i = 0x5f3759df - (i >> 1);  
    float y; memcpy(&y,&i,4); // approximate  
    return y * (1.5f - 0.5f * x * y * y); // better  
}
```

Compilers use them at Ofast  
Accuracy 2ULP  
Non standard: Hardware dependent

Real x86\_64 code for  $1/\sqrt{x}$

```
_mm_store_ss( &y, _mm_rsqrt_ss( _mm_load_ss( &x ) ) );  
return y * (1.5f - 0.5f * x * y * y); // One round of Newton's method
```

Real x86\_64 code for  $1/x$

```
_mm_store_ss( &y, _mm_rcp_ss( _mm_load_ss( &x ) ) );  
return (y+y) - x*y*y; // One round of Newton's method
```

# Cost of functions (in cpu cycles i7sb)

	Gnu libm	Cephes scalar	Cephes autovect	Cephes handvect	Approx (16bits)	Intel svml	Amd libm
	s d	s d	s d	s		s d	s d
sin,cos large x	55 100 <b>&gt;500</b>	30 50	11 30	20		12 30	25 45
sincos	70	40	15	22			50
atan2	50 100	30	13			17 52	67 87
exp	<b>650</b> 65	42 55	10 23	27		12 26	16 36
log	50 105	37 42	11 28	24	12	12 30	27 59
SET_RESTORE_ROUND_NOEXF (FE_TONEAREST);							

# How to speed up math

- Avoid or factorize-out division and sqrt
  - if possible compile with “-Ofast”
- Prefer linear algebra to trigonometric functions
- Cache quantities often used
  - No free lunch: at best trading memory for cpu
- Choose precision to match required accuracy
  - Square and square-root decrease precision
  - Catastrophic precision-loss in the subtraction of almost-equal large numbers

# Example: cut in pt, phi, eta

```
inline float pt2(float x, float y) {return x*x+y*y;}
inline float pt(float x, float y) {return std::sqrt(pt2(x,y));}
inline float phi(float x, float y) {return std::atan2(y,x);}
inline eta(float x, float y, float z) { float t(z/pt(x,y)); return ::asinhf(t);}
inline float dot(float x1, float y1, float x2, float y2) {return x1*x2+y1*y2;}
inline float dphi(float p1,float p2) {
    auto dp=std::abs(p1-p2); if (dp>float(M_PI)) dp-=float(2*M_PI);
    return std::abs(dp);
};
```

```
if (pt(x[i],y[i])>ptcut) ...
```

```
if (dphi(phi(x[i],y[i]),phi(x[j],y[j]))<phicut) ....
```

Exercise in ptcut.cpp

# Formula Translation:

what was the author's intention?

```
// Energy loss and variance according to Bethe and Heitler, see also
// Comp. Phys. Comm. 79 (1994) 157.
//
double p = mom.mag();
double normalisedPath = fabs(p/mom.z())*radLen;
double z = exp(-normalisedPath);
double varz = (exp(-normalisedPath*log(3.)/log(2.))- exp(-2*normalisedPath));
```

```
double pt = mom.perp();
double j = a_i*(d_x * mom.x() + d_y * mom.y()/(pt*pt);
double r_x = d_x - 2* mom.x()*(d_x*mom.x()+d_y*mom.y()/(pt*pt);
double r_y = d_y - 2* mom.y()*(d_x*mom.x()+d_y*mom.y()/(pt*pt);
double s = 1/(pt*pt*sqrt(1 - j*j));
```

Exercise: edit Optimizelt.cc to make it “optimal”

check the generated assembly, modify Benchmark.cpp to verify speed gain

# Formula Translation: the solution?

Is the compiler able to do it for us?

# PRECISION, ACCURACY, SPEED

# Definitions (mine)

## ■ Precision

- ❑ Numerical precision (in bits)
- ❑ Can be evaluated using a higher “precision”

## ■ Accuracy

- ❑ The error w/r/t the truth
- ❑ Can be evaluated comparing different algorithms

## ■ Target Accuracy

- ❑ The acceptable tolerance w/r/t the above
- ❑ Evaluated, for instance, w/r/t know error on the truth



# Precision, accuracy, speed

Quadratic equation

$$ax^2 + bx + c = 0$$

Compare the “formula translated” code with an optimized one for accuracy (look in wikipedia)

Find the fastest algorithm (for single precision arguments) covering a given range of a,b,c with “good-enough” accuracy

Repeat for vector code

# Example: multiple scattering formula

```
double ms(double radLen, double m2, double p2) {  
    constexpr double amscon = 1.8496e-4; // (13.6MeV)**2  
    double e2    = p2 + m2;  
    double beta2  = p2/e2;  
    double fact = 1 + 0.038*log(radLen); fact *=fact;  
    double a = fact/(beta2*p2);  
    return amscon*radLen*a;  
}
```

Already an  
approximation

Material density,  
thickness, track angle  
Known at percent?

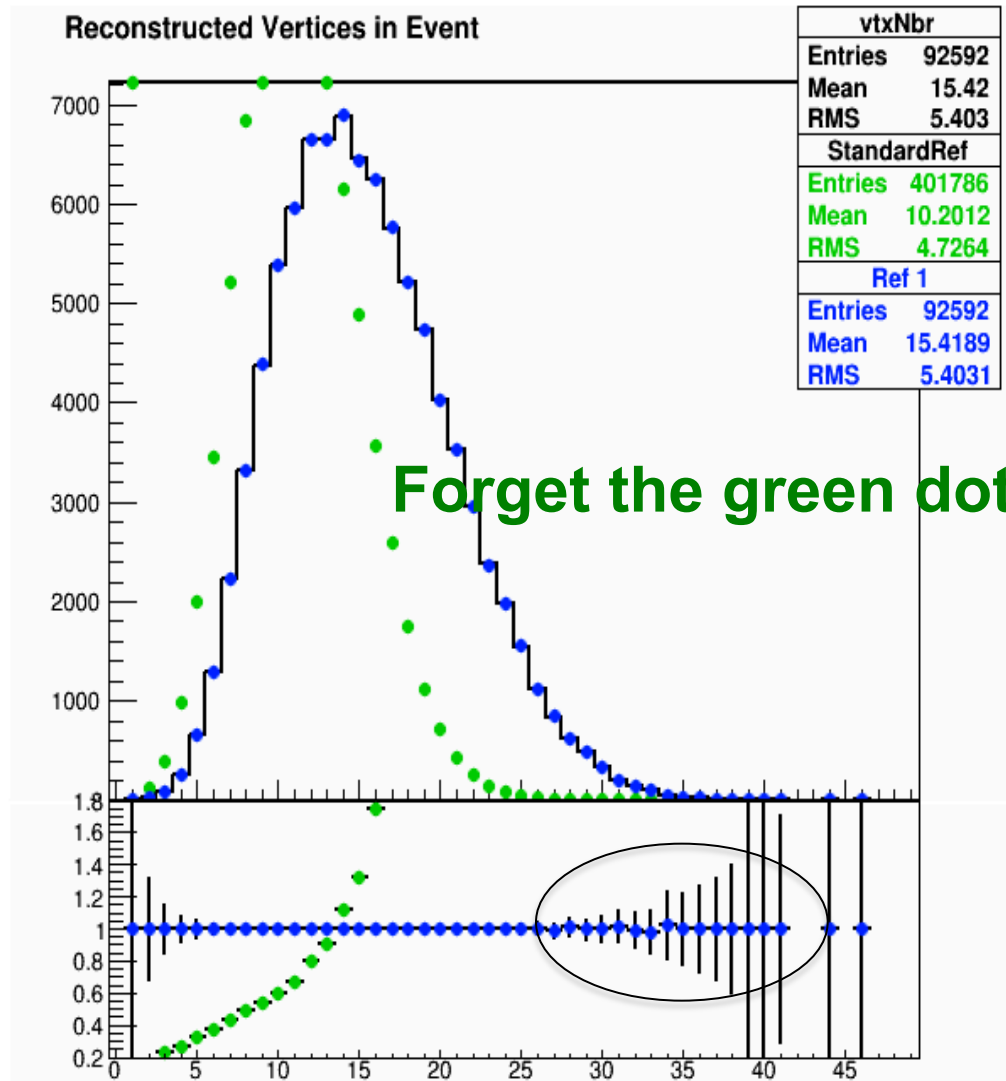
```
float msf(float radLen, float m2, float p2) {  
    constexpr float amscon = 1.8496e-4; // (13.6MeV)**2  
    float e2    = p2 + m2;  
  
    float fact = 1.f + 0.038f*unsafe_logf<2>(radLen); fact /= p2;  
    fact *=fact;  
    float a = e2*fact;  
    return amscon*radLen*a;  
}
```

2<sup>nd</sup> order polynomial

# Anedoct: regression in CMS validation

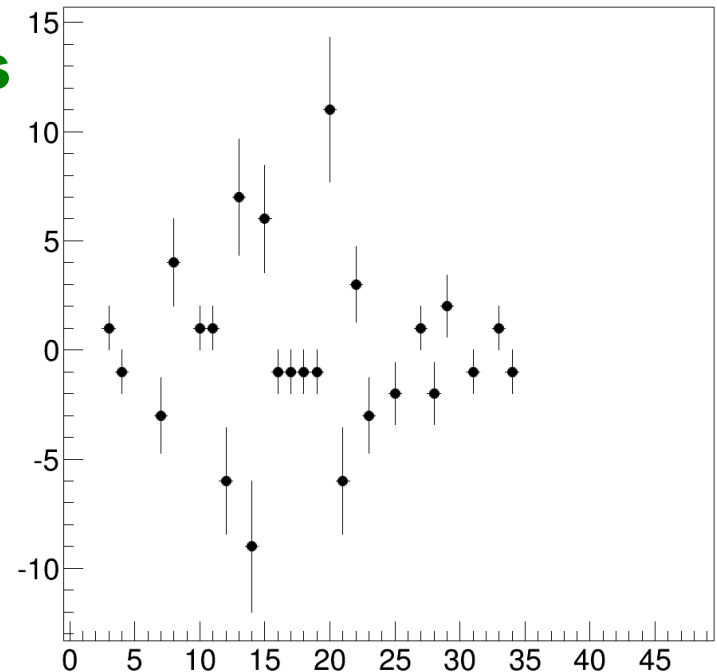
- 192 events out of 92592 had a different number of vertices (number of tracks, number of hits per tracks) for two identical relval productions
- Running 400 times the same event on the CERN cluster it was found that the difference was related to the machine where the job landed
- Eventually discovered that the discriminant was **AMD vs INTEL**

# Evidence



Running twice the very same  
Release validations  
Differences in all bins...

Diff Primary Vertex Collection Size



# AMD vs Intel

- Fast reciprocal is not standard
  - Easy to reproduce ( $1/\sqrt{0.1}$ )

```
cat oneLineRecip.cpp
#include <cmath>
#include <cstdio>

int main(int n, char * v[]) {
    float fn = n - 1.f;
    float k = 0.1f + fn;

    float q = 1.f/std::sqrt(k);
    printf("%a %a\n",k,q);
    return 0;
}
```

## AMD

```
c++ -Ofast oneLineRecip.cpp; ./a.out
0x1.9999ap-4 0x1.94c582p+1
```

## INTEL

```
./a.out # same binary!
0x1.9999ap-4 0x1.94c58p+1
```

# THE EXERCISE: N-Body problem

- N (large!) interacting particle in a Box (cube)
  - Simulation using discrete steps in time
  - $F = ma$ ;  $\delta v = a\delta t$ ;  $\delta x = v\delta t$
  - $a = F/m$ ;  $v += a\Delta t$ ;  $x += v\Delta t + a\Delta t^2/2$
  - In 3D! (assume an arrow above  $F, a, v, x$ )
- F is Coulomb/Gravity:  $1/r^2$ 
  - Variations: longer/shorter range
  - All attractive, all repulsive or mix charges
- Walls: elastic scattering
- Challenge: window..

# Numerical issues

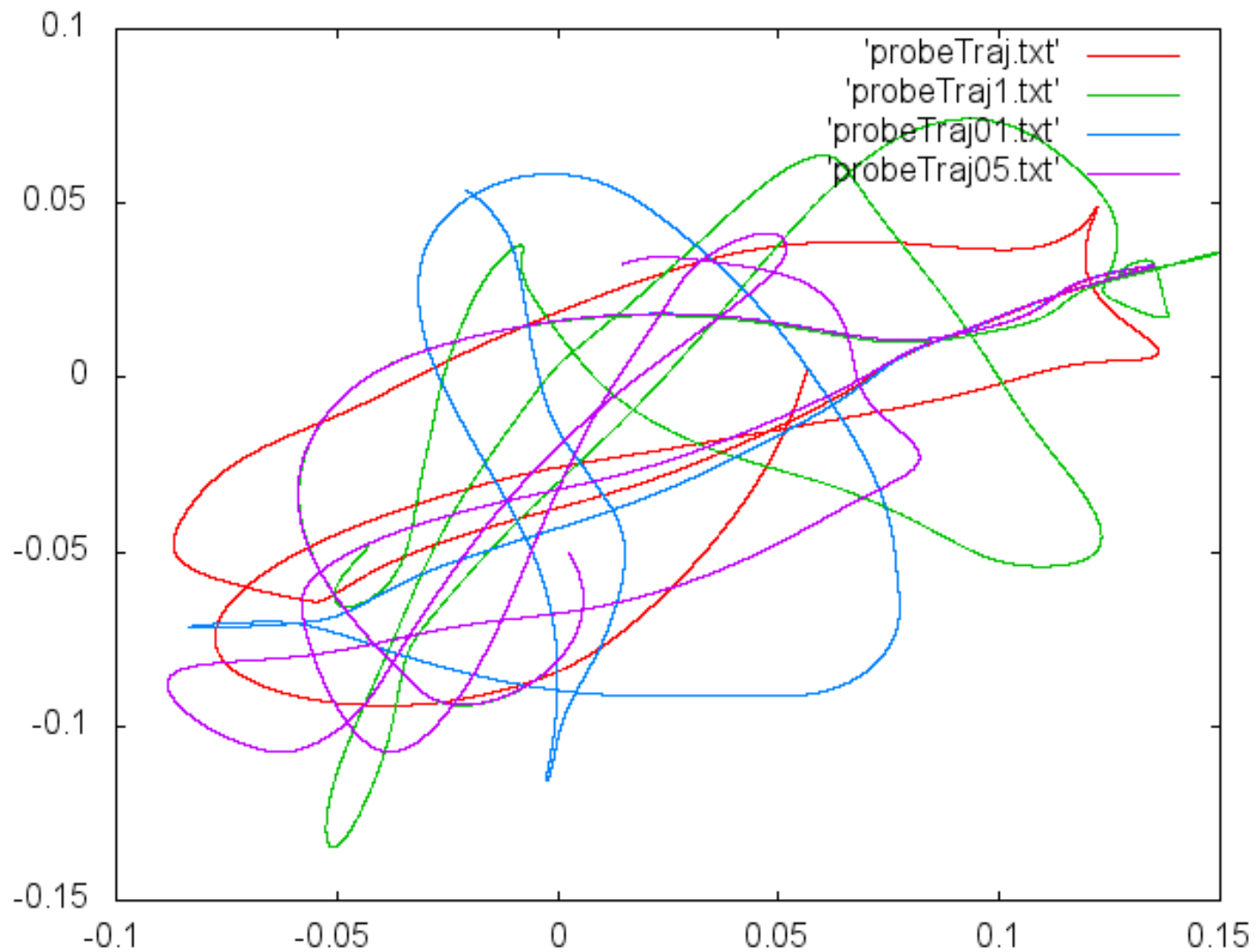
- What may limit precision, accuracy?
- What will determine the target accuracy?
- What to “measure”?
- How many free parameters the problem have?
- Challenge:
  - Fast and accurate

# Step1

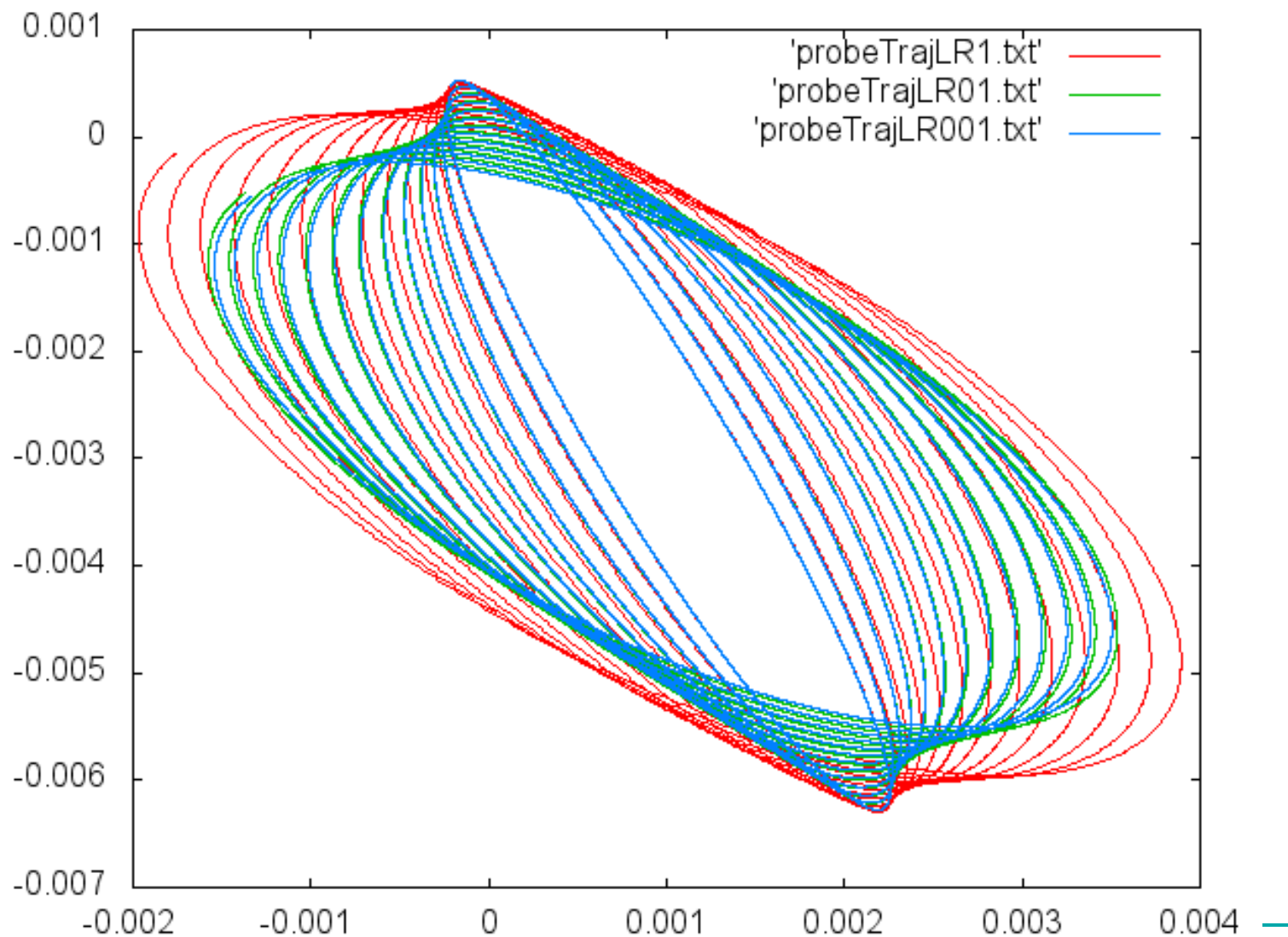
- Force on a single particle
  - All other  $N-1$  particles do not move
- What I give you:
  - A simple **Vector3D** class
  - A **Particle** class
  - A “main template”
  - **Feel free to modify them following your needs**



# Result $1/r^2$



# Result 1/r



# Step2

## ■ Bouncing Box

- Add walls of a box and let the particles scatter on them
- To test: do not apply any force

# Step3

- N interacting particles

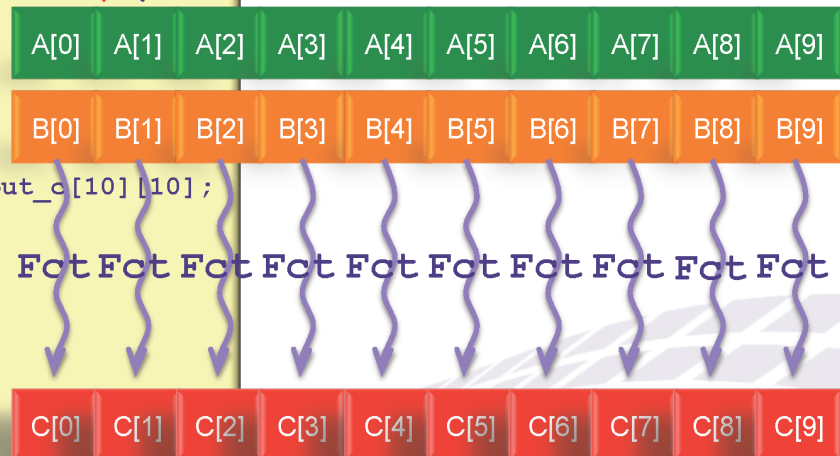
# SIMD: LOOP VECTORIZATION

# What is Stream Computing?

- A similar computation is performed on a collection of data (*stream*)
  - There is no data dependence between the computation on different stream elements
- Stream programming is well suited to GPU *and vector-cpu!*

```
kernel void Fct(float a<>, float b<>, out float c<>) {  
    c = a + b;  
}  
  
int main(int argc, char** argv) {  
    int i, j;  
    float a<10, 10>, b<10, 10>, c<10, 10>;  
    float input_a[10][10], input_b[10][10], input_c[10][10];  
    for(i=0; i<10; i++) {  
        for(j=0; j<10; j++) {  
            input_a[i][j] = (float) i;  
            input_b[i][j] = (float) j;  
        }  
    }  
    streamRead(a, input_a);  
    streamRead(b, input_b);  
    Fct(a, b, c);  
    streamWrite(c, input_c);  
    ...  
}
```

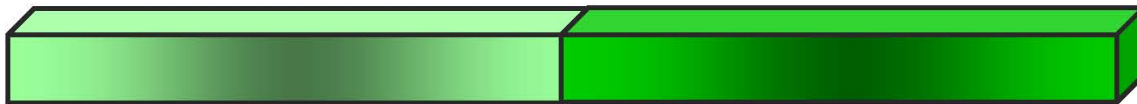
Brook+ example



# SSE Data types



**4x floats**



**2x doubles**



**16x bytes**



**8x 16-bit shorts**



**4x 32-bit integers**



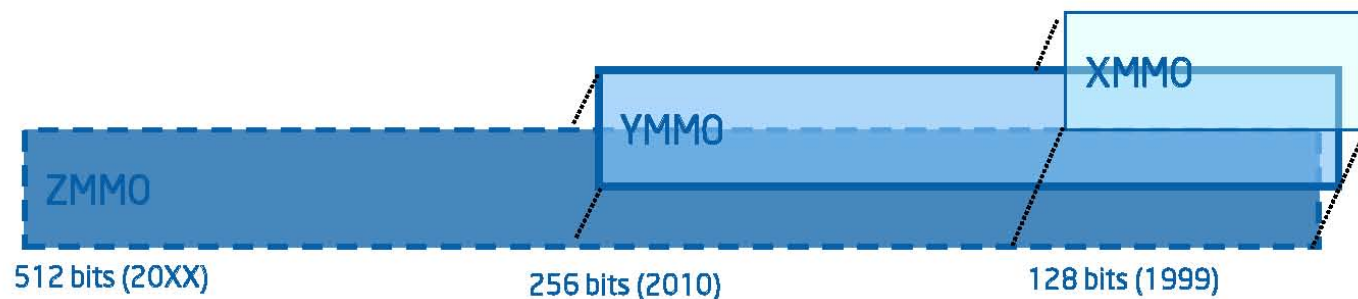
**2x 64-bit integers**



# Sandy Bridge (2010): Intel® AVX

*A 256-bit vector extension to SSE*

- Intel® AVX extends all 16 XMM registers to 256bits
- Intel® AVX works on either
  - The whole 256-bits
  - The lower 128-bits (like existing SSE instructions)
    - A drop-in replacement for all existing scalar/128-bit SSE instructions
- The new state extends/overlays SSE
- The lower part (bits 0-127) of the YMM registers is mapped onto XMM registers





# Single Instruction Multiple Data

- SLP (Superword Level Parallelism)
  - Direct mapping to underlying SIMD machine instruction
  - Usually implemented using array/vector notation
- Loop Vectorization
  - Transform a loop into  $N$  streams ( $N$ =SIMD-width)
  - Compiler assisted or implemented in a “vector-library”
- Loop vectorization is more efficient than SLP
  - Transform your problem in a long loop over simple quantities

# “Vector Extension”

- SIMD vector can be found implemented as compiler's extension or libraries
  - <https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html>
  - <http://clang.llvm.org/docs/LanguageExtensions.html#vectors-and-extended-vectors>
  - <http://www.cilkplus.org/tutorial-array-notation>
  - <http://code.compeng.uni-frankfurt.de/projects/vc>
- Here we will experiment with “gcc vectors”

# Gcc Vector extension

```
typedef T __attribute__((vector_size( N*sizeof(T) ))) VecNT;
```

```
Vec4F a{0,1f,-2f,3f}, b{-1f,-2f,3f,0}, c{0,2.f,4.f,5.f};
```


```
c += 3.14f*b*a;
```

```
auto z = (a>0) ? a : -a;
```

```
auto m = (a>b) ? a : b;
```

```
auto t = a/b; t = (x>pi/8.f) ? (t-1.0f)/(t+1.0f) : t;
```

**Vector of integers:**  
**0 for false, -1 for true**



**always computed**



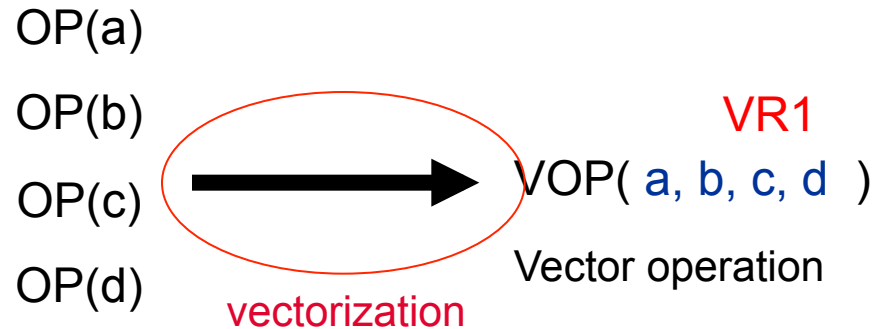
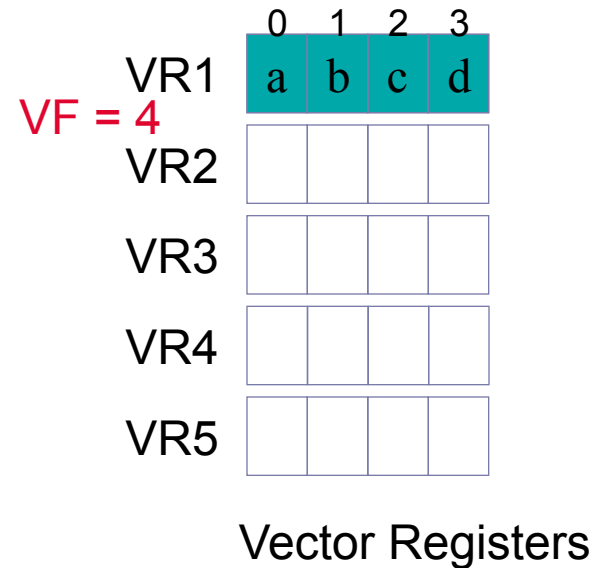
# How to use SIMD vectors?

- In place of Vector3D
  - ❑ Works
  - ❑ Waste 1/4 of memory if used in storage
  - ❑ Not useful in code dominated by operations among elements of the same vector: `phi()`, `perp()`...
  - ❑ Limited to  $N=4$
- In place of T
  - ❑ “Easy” to make any algorithm based on T to work for VecNT
  - ❑ Up to the user to split the problem in chunks of size N
- Let the compiler do it for us

# “Auto” vectorization

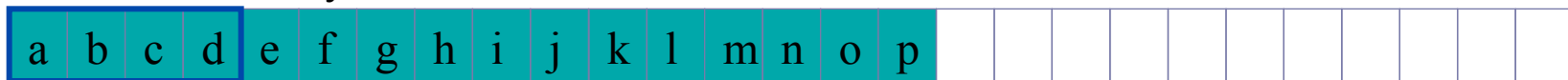
- The process that transform scalar code in vector code
- As a compiler pass
  - Issues: detect and manage data dependencies
  - Hints from user as macro
- OpenMP4
  - New: implementations still experimental
  - icc: “fully supported?”
  - gcc: syntax and code generation ok, vectorization relies on the corresponding compiler pass
- OpenCL:
  - See Tim’s lectures

# What is vectorization?



- Data elements packed into vectors
- Vector length  $\rightarrow$  Vectorization Factor (VF)

Data in Memory:



# Vectorization

More at <http://gcc.gnu.org/projects/tree-ssa/vectorization.html>

◇ original serial loop:

```
for(i=0; i<N; i++){  
    a[i] = a[i] + b[i];  
}
```

vectorization



◇ loop in vector notation:

```
for (i=0; i<(N-N%VF); i+=VF){  
    a[i:i+VF] = a[i:i+VF] + b[i:i+VF];  
}
```

vectorized loop

```
for ( ; i < N; i++) {  
    a[i] = a[i] + b[i];  
}
```

epilog loop

◇ loop in vector notation:

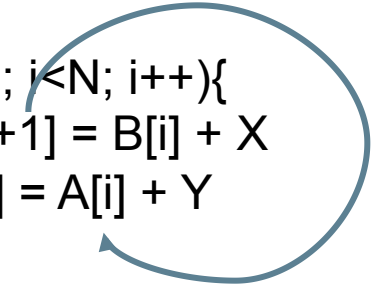
```
for (i=0; i<N; i+=VF){  
    a[i:i+VF] = a[i:i+VF] + b[i:i+VF];  
}
```

◇ Loop based vectorization

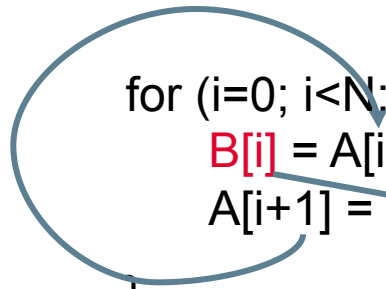
◇ No dependences between iterations

# Loop Dependence Tests

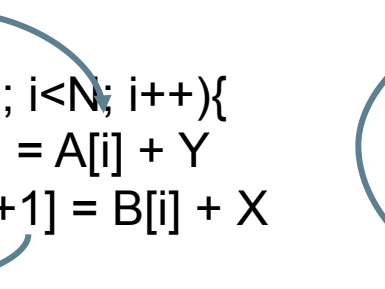
```
for (i=0; i<N; i++){  
    A[i+1] = B[i] + X  
    D[i] = A[i] + Y  
}
```



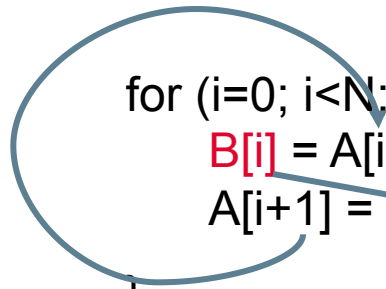
```
for (i=0; i<N; i++){  
    A[i+1] = B[i] + X  
    for (i=0; i<N; i++){  
        D[i] = A[i] + Y  
    }  
}
```



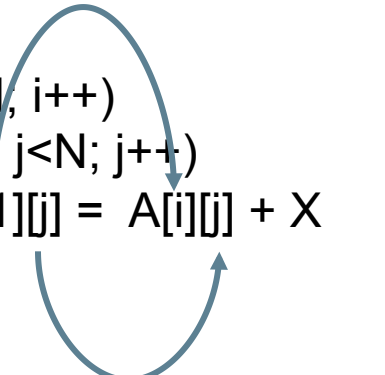
```
for (i=0; i<N; i++){  
    B[i] = A[i] + Y  
    A[i+1] = B[i] + X  
}
```



```
for (i=0; i<N; i++){  
    B[i] = A[i] + Y  
    A[i+1] = B[i] + X  
}
```



```
for (i=0; i<N; i++){  
    for (j=0; j<N; j++){  
        A[i+1][j] = A[i][j] + X  
    }  
}
```



Subtle issues:

**A** may partially overlap  
with **B**  
with **X**  
with **N**

The compiler will generate  
runtime checks and alternative  
sequential code.

To avoid this overhead  
use local variable,

“restrict” keyword or  
**#pragma GCC ivdep**

```
void ignore_vec_dep (int *a, int k, int c, int m)  
{  
    #pragma GCC ivdep  
    for (int i = 0; i < m; i++)  
        a[i] = a[i + k] * c;  
}
```



# Reduction

```
float innerProduct() {  
    float s=0;  
    for (int i=0; i!=N; i++)  
        s+= a[i]*b[i];  
    return s;  
}
```

loop in vector notation:

```
for (i=0; i<N; i+=VF){  
    s[0:VF] += a[i:i+VF] * b[i:i+VF];  
}  
return hsum(s);
```

```
// pseudo code  
float innerProduct() {  
    float s[VF]={0};  
    for (int i=0; i!=N; i+=VF)  
        for (int j=0;j!=VF;++j)  
            s[j]+=a[i+j]*b[i+j];  
    // horizontal sum;  
    float sum=s[0];  
    for (int j=1;j!=VF;++j)sum+=s[j];  
    return sum;  
}
```

Requires *relaxed* float-math rules

# Conditional code

```
void foo() {  
    for (int i=0; i!=N; i++) {  
        if (b[i]>a[i]) c[i]=a[i]*b[i];  
        else c[i]=a[i];  
    }  
}
```

loop in vector notation:

```
for (i=0; i<N; i+=VF){  
    t[0:VF] = b[i:i+VF] > a[i:i+VF];  
    // compute both branches  
    x[0:VF] = a[i:i+VF] * b[i:i+VF];  
    y[0:VF] = a[i:i+VF];  
    // mask and “blend”  
    c[i:i+VF] = t&x | !t&y;  
}
```

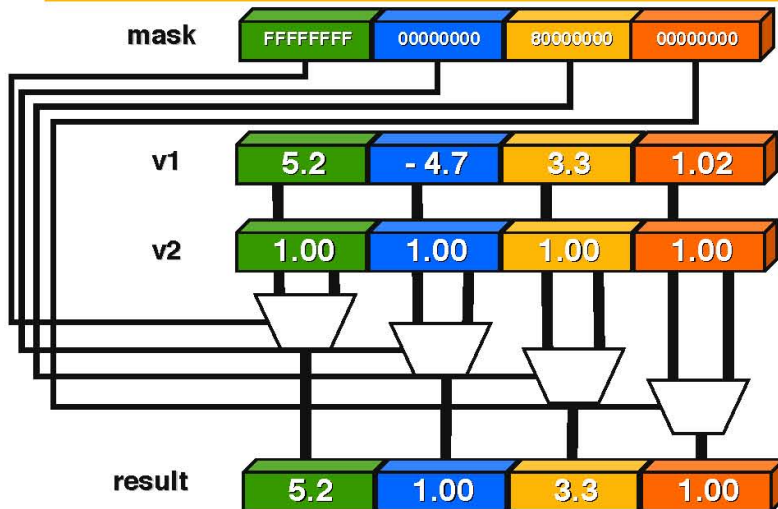
```
// pseudo code  
void foo() {  
    for (int i=0; i!=N; i++) {  
        //evaluate condition  
        int t = b[i]>a[i] ? ~0 : 0;  
        // compute both branches  
        float x= a[i]*b[i];  
        float y=a[i];  
        // mask and “blend”  
        c[i] = t&x | ~t&y;  
    }  
}
```

**The compiler is able to make the transformation of a condition in “*compute, mask and blend*” if code is not too complex**

## Blends: To Boost Conditionals SIMD flows

**SSE 4, AVX  
only**

```
/*Integer blend instructions */
_mm_blend_epi16 (__m128i v1, __m128i v2, const int mask);
_mm_blendv_epi8 (__m128i v1, __m128i v2, __m128i mask);
/*Float single precision blend instructions */
_mm_blend_ps (__m128 v1, __m128 v2, const int mask);
_mm_blendv_ps (__m128 v1, __m128 v2, __m128 v3);
/*Float double precision blend instructions */
_mm_blend_pd (__m128d v1, __m128d v2, const int mask);
_mm_blendv_pd (__m128d v1, __m128d v2, __m128d v3);
```



### •Used to code conditional SIMD flows

```
for (i=0; i<N; i++)
    if (a[i]<b[i]) c[i]=a[i]*b[i];
    else c[i]=a[i];
```

### Vector code assuming:

```
for (i=0; i< N; i+=4){
    A = _mm_loadu_ps(&a[i]);
    B = _mm_loadu_ps(&b[i]);
    C = _mm_mul_ps (A, B);
    mask = _mm_cmplt_ps (A, B);
    C = _mm_blend_ps (C, A, mask);
    _mm_storeu_ps (&c[i], C);
}
```



# Managing “rare” divergences

```
auto t = a/b; t = (x>pi/8.f) ? (t-1.0f)/(t+1.0f) : t;
```

- ❑ What about if  $x$  is often  $< \pi/8$  ?
- OpenCL (and CUDA and Altivec) provides two functions
  - ❑ **all** / **any** that return true if **all elements** / **at least one of the element** of the vector are/is non zero
  - ❑ in SSE/AVX can be constructed using “movmsk” instruction
  - ❑ In Neon: more involved...
  - ❑ (see NativeVector.h)

# Exercise 1

- Open SimpleVectorization.cc or pi.cc
  - or <http://goo.gl/SAiici> / <http://goo.gl/Vwvlq3>
- Compile it with O3, Ofast, more fancy options
  - Analyze the compiler report
    - Correlate with code and generated instruction
    - Is vectorizing everything?
    - (if too complex, comment-out all but one function)
    - Add " -fopt-info-vec"
    - Try "-fopt-info-vec-missed -fno-tree-slp-vectorize"

*objdump -S -r -C --no-show-raw-insn -w xyz.o | less*

*build SimpleVectorization\_vect.s;*

*cat SimpleVectorization\_vect.s | less*

## Exercise 2

- The Mandelbrot set is defined to be that set of points  $c$  such that the iteration  $z = z^2 + c$  does not escape to infinity, with  $z$  initialized to 0.
- Brute force approach to compute area: sample circle, count number of point not escaping
  - (see <https://www.fractalus.com/kerry/articles/area/mandelbrot-area.html>)
- Take mandel.cpp, vectorize using NativeVector.h

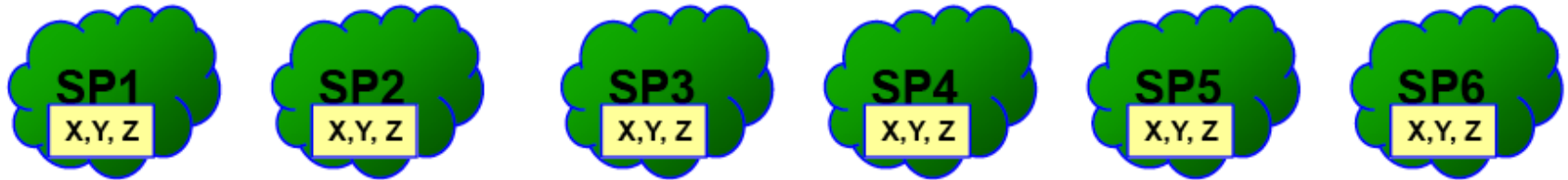
---

[https://github.com/CppCon/CppCon2014/tree/master/  
Presentations/Data-Oriented%20Design%20and%20C%2B%2B](https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B)

# DATA ORGANIZATION

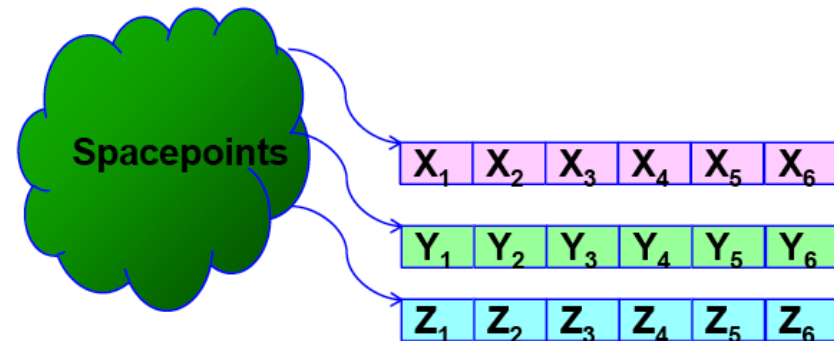
# Data Organization: AoS vs SoA

- Traditional Object organization is an Array of Structure
  - Abstraction often used to hide implementation details at object level



- Difficult to fit stream computing
- Better to use a Structure of Arrays

- OO can wrap SoA as the AoS
  - Move abstraction higher
  - Expose data layout to the compiler



- Explicit copy in many cases more efficient
  - (notebooks vs whiteboard)



# A simple SOA binding

```
Vector3D<float> va{-3, 0, 2.};           // standard local vectors
Vector3D<float> vb{1, 2, 3.14};

float x[5]={1.f}, y[5]={1.f}, z[5]={1.f}; // sort of SoA

Vector3D<float const> v1(x[0],y[0],z[0]); // right hand
binding
Vector3D<float> v5(x[4],y[4],z[4]); // left hand binding

v5 = v1+vb-va; // just works!
```

- SoA can be implemented as a hierarchy of `std::tuple` of `std::vector`
  - matching the *natural* data-model hierarchy
- binding through the constructors
- Here we do it “by hand”

# SOA for Vector3D

```
template<typename T> using AVector = std::vector<T,align_allocator<T,32>>;
```

```
template<typename T>
```

```
class SOA3D {
```

```
public:
```

```
    using V3D = Vector3D<T>;
```

```
    using R3D = Vector3D<T&>;
```

```
    using C3D = Vector3D<T const&>;
```

```
    SOA3D(){}
```

```
    explicit SOA3D(unsigned int is) : vx(is),vy(is),vz(is){}
```

```
    void resize(unsigned int n) { vx.resize(n);vy.resize(n);vz.resize(n);}
```

```
    R3D operator[](unsigned int i) { return R3D(vx[i],vy[i],vz[i]); }
```

```
    C3D operator[](unsigned int i) const { return C3D(vx[i],vy[i],vz[i]); }
```

```
    unsigned int size() const { return vx.size();}
```

```
private:
```

```
    AVector<T> vx,vy,vz;
```

```
};
```

# SoA for Particles

```
template<typename T>
class Particles {
public:
    using Float = T;
    using CP = Particle<Float const &>;
    using RP = Particle<Float &&>;
    using LP = Particle<Float&>;
    using Soa = SOA3D<Float>;
    using uint = unsigned int;

    Particles(){}
    explicit Particles(uint n) : m_pos(n), m_vel(n), m_acc(n), m_mass(n), m_charge(n), m_n(n){}

    void resize(uint n) { m_pos.resize(n); m_vel.resize(n); m_acc.resize(n); m_mass.resize(n); m_charge.resize(n); m_n=n;}
    uint size() const { return m_n;}

    CP operator[](uint i) const { return CP(m_mass[i], m_pos[i], m_vel[i], m_acc[i]); }
    LP operator[](uint i)      { return LP(m_mass[i], m_pos[i], m_vel[i], m_acc[i]); }

private:
    Soa m_pos;
    Soa m_vel;
    Soa m_acc;
    AVector<Float> m_mass;
    AVector<Float> m_charge;
    uint m_n;
};
```

# Vectorization of “math function”

- Exploit compiler + vendor libraries
  - Requires licensed libs by intel and/or amd
  - No change in user code: just recompile
    - `-mveclibabi=svml -L/.../lib/intel64 -lsvml -lirc`
    - `-mveclibabi=acml -L/.../lib/amd64 -lacml -lamdlibm`
- Exploit auto-vectorization
  - Modified cephess library (or other open source)
    - Requires header-file + different function names
  - Look into vdt/

# Don't stop the stream!

- Vector code is effective as long as
  - We do not go back to memory
    - Operate on local registries as long as possible
  - We maximize the number of useful operations per cycle
    - Conditional code is a killer!
    - Better to compute all branches and then blend
- Algorithms optimized for sequential code are not necessarily still the fastest in vector
  - Often slower (older...) algorithms perform better

# ADVANCED TOPICS

# Example: Gaussian random generator

- The fastest (scalar) method to produce random number following a normal (Gaussian) distribution is the *ziggurat method* by Marsaglia
  - split the pdf in rectangles and use a look-up method
  - In 2% of the cases it needs more computation (and rejections)
- The traditional algorithm is the Box–Muller method
  - that throws two independent random numbers  $U$  and  $V$  distributed uniformly on  $(0,1]$ . The two following random variables  $X$  and  $Y$  will be normal distributed

$$X = \sqrt{-2 \ln U} \cos(2\pi V),$$
$$Y = \sqrt{-2 \ln U} \sin(2\pi V).$$

# Example: Gaussian random generator

- The Polar method, due to Marsaglia, is often used
  - In this method  $U$  and  $V$  are the coordinate of a point inside a circle of radius 1 obtained drawing them from the uniform  $(-1, 1)$  distribution, and then  $S = U^2 + V^2$  is computed. If  $S$  is greater or equal to one then the method starts over, otherwise the following two quantities, normal distributed, are returned

$$X = U \sqrt{\frac{-2 \ln S}{S}}, \quad Y = V \sqrt{\frac{-2 \ln S}{S}}$$

Code from gcc libstdc++ (bits/random.tcc)

```
result_type __x, __y, __r2;
do
{
    __x = result_type(2.0) * __aurng() - 1.0;
    __y = result_type(2.0) * __aurng() - 1.0;
    __r2 = __x * __x + __y * __y;
}
while (__r2 > 1.0 || __r2 == 0.0); // rejection 14% of the time

const result_type __mult = std::sqrt(-2 * std::log(__r2) / __r2);
```



# Speed of operations

**GHz**

Floating Point Operations, cache access

Branches, function calls,

User  
time

**MHz**

Memory access, virtual calls, malloc

System  
time

**KHz**

malloc, I/O, IPC

Wait

**Hz**

Services, database,

# Cash-Karp Runge-Kutta Step

## 3. A STRATEGY FOR DEALING WITH NONSMOOTH BEHAVIOR

The Runge-Kutta formula derived in the previous section has the special property that it contains imbedded solutions of all orders less than five. In addition, the formula has been designed so that the first five  $c_i$  values span the range  $[0, 1]$  with reasonable uniformity, so that we have a very good chance of spotting bad behavior in  $f$  if it occurs. Our aim is to derive an automatic strategy that allows us to quit early, i.e., before all six function evaluations have been computed on the current step, if we suspect trouble, and to accept a lower order solution if appropriate.

We assume that we have computed a numerical solution  $y_{n-1}$  at the step point  $x_{n-1}$  and that for the current step, from  $x_{n-1}$  to  $x_n = x_{n-1} + h$ , all six function evaluations are computed so that solutions of all orders from 1 to 5 are available. (We guarantee this situation for the first step with  $n = 1$ ). We denote the imbedded solution of order  $i$  at  $x_n$  by  $y_n^{(i)}$ ,  $1 \leq i \leq 5$ , and define

$$\text{ERR}(n, i) = \|y_n^{(i+1)} - y_n^{(i)}\|^{1/(i+1)}, \quad \text{for } i \in 1, 2, 4. \quad (6)$$

We exclude the case  $i = 3$  for two reasons. First, following the approach of Shampine et al. [15], we allow only a few different orders to be used, and we have chosen to allow orders 2, 3, or 5. Second,  $\text{ERR}(n, 3)$  is of no use in predicting when to quit early since all six  $k_i$ 's are required before  $y_n^{(4)}$  can be computed.

Suppose now that we were to accept the solution of order 5 at  $x_n$ . We wish to compute a suitable step length,  $\bar{h}_4$ , to be used in integrating from  $x_n$  to  $x_{n+1}$  using a 5(4) formula. A typical step-choosing strategy would compute  $\bar{h}_4$  as

$$\bar{h}_4 = \frac{\text{SF} \times h}{E(n, 4)}, \quad \text{where } E(n, 4) = \frac{\text{ERR}(n, 4)}{\epsilon^{1/5}}. \quad (7)$$

Here  $\epsilon$  is the local accuracy required (as specified by the user) and SF is a safety factor often taken to be 0.9. Similarly, if we were to accept either the second- or third-order solution at  $x_n$ , the steplengths  $\bar{h}_1$ ,  $\bar{h}_2$ , respectively, that would be selected at the next step by our step-control algorithm would be

$$\bar{h}_i = \frac{\text{SF} \times h}{E(n, i)}, \quad \text{where } E(n, i) = \frac{\text{ERR}(n, i)}{\epsilon^{1/(i+1)}}, \quad i \in 1, 2. \quad (8)$$

0.9\*step/pow(err/eps,0.2)

# Summary of Exercises

- Count the number of “floats” between 100 and 101
- Change rounding (in patriot.cpp)
- Use Kahan summation, measure speed (in patriot.cpp)
  - Compare with double precision
  - Try to vectorize
- Improve performance of code in slide 29
- Estimate accuracy required in Energy Loss
  - Try to use an approximate exp, log
  - Measure speed
- solve quadratic equation in optimal way
  - Ask help to wikipedia
  - Measure speed and accuracy of various approach
  - Vectorize