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 excercises

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

which c++-482

c++-482 -v

cd FPOptimization/exercises

make afloat

./afloat_O2

./afloat_fast

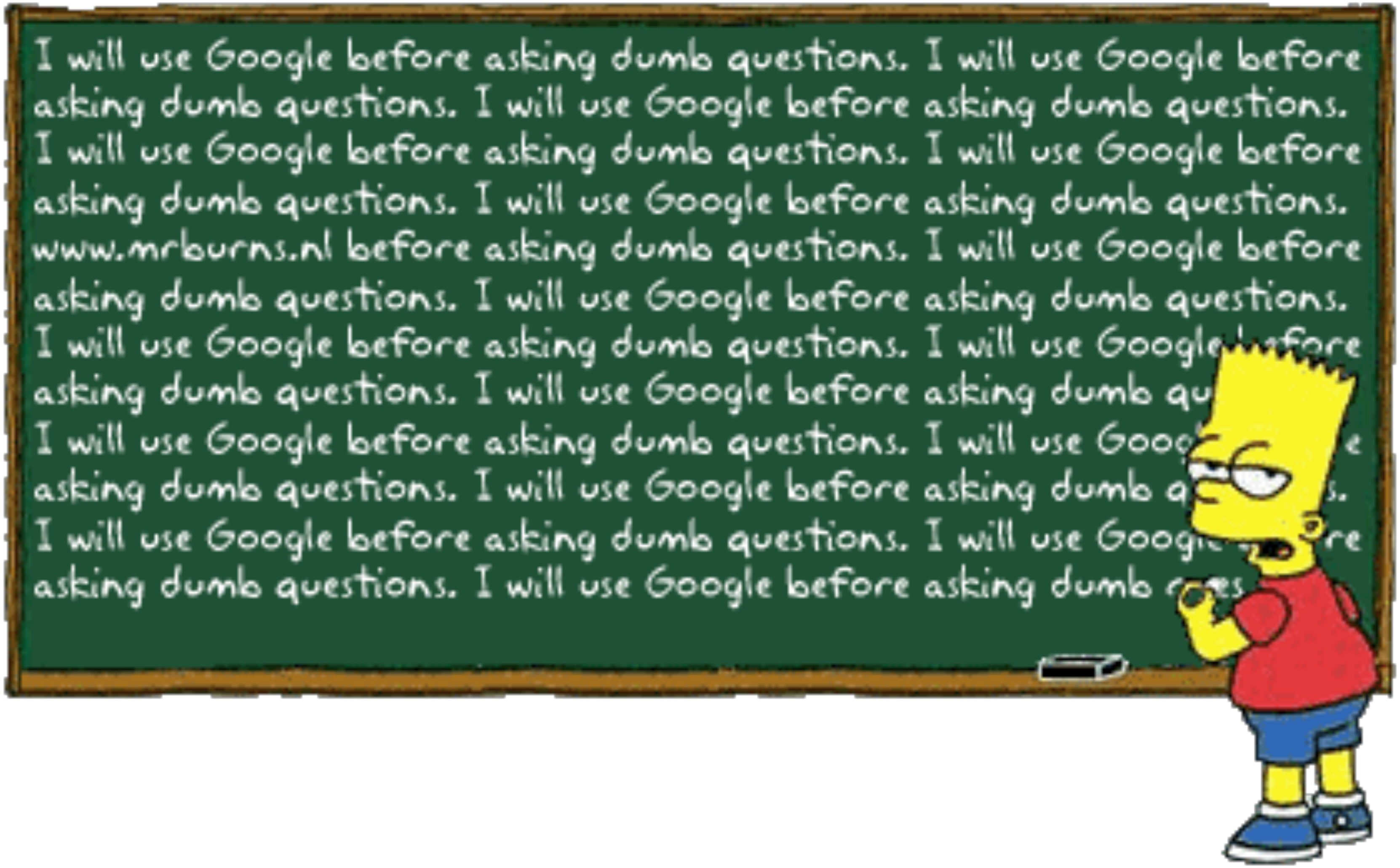
ldd ./afloat_O2 : check

libstdc++.so.6 => /opt/gcc482/lib64/libstdc++.so.6

Disclaimer, caveats, references

- This is NOT a full overview of IEEE 754
- It applies to x86_64 systems and gcc > 4.5.0
  - 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

Don’t be afraid to ask questions!



Floating behaviour…

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


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;
}
```

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

```
make intLoop
make floatLoop
./intLoop_O; ./floatLoop_O2;
```

```
2147483647 -2147483648
16777216: 1.67772e+07 1.67772e+07 1.67772e+07
0x1.fffffep+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: 1.4×10^{-45} to 3.4×10^{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
Make patriot
./patriot_O2

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
```

Floating behaviour…

```
void fcos() {
 float pi2=2.f*std::acos(-1.f);
 float x=1.f;
 while (std::cos(x)==std::cos(x+pi2))
   x+=pi2;
}

  x=1; float y=x;
 while (std::abs(std::cos(x)-std::cos(x+pi2))<1.e-4f) {
   y=x; x+=pi2;
}

  x=1;
  while (fabs(std::cos(1.f)-std::cos(x+pi2))<1.e-4f)
    x+=pi2;
```

For which value of x these loops will stop? Why?

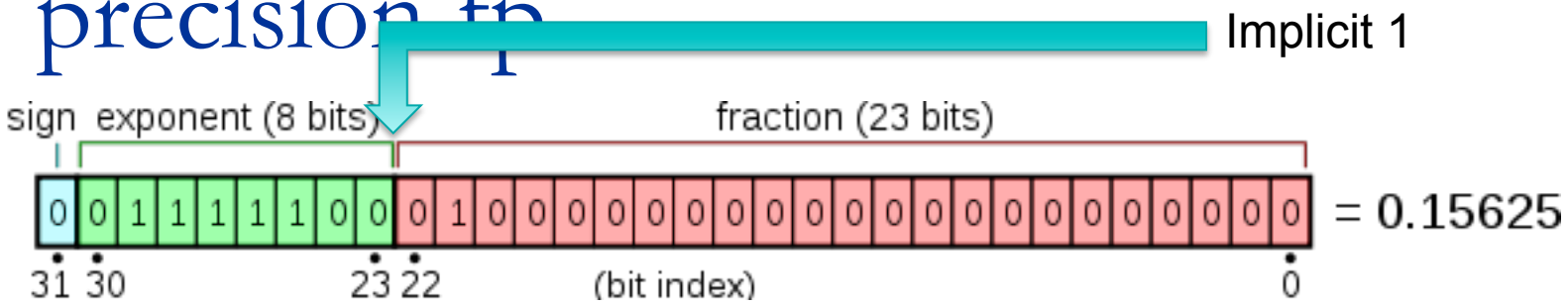
```
cat fcos.cpp
make fcos
./fcos_O2

  6.28319
  0.540302

  32761.6 32767.9
  0.461628 0.463344

  346.575
  0.540402 0.540413
```

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_H	zero, −0	subnormal numbers	$(-1)^{signbits} \times 2^{-126} \times$ 0.significandbits
01_H, ..., FE_H	normalized value	normalized value	$(-1^{signbits} \times 2^{exponentbits-127}$ × 1.significandbits
FF_H	±infinity	NaN (quiet,signalling)	

Look into a float

```
void look(float x) {
  printf("%a\n",x);
  int e; float r = ::frexpf(x,&e);
  std::cout << x << " exp " << e << " res " << r << std::endl;

  union {
    float val;
    int bin;
  } f;

  f.val = x;
  printf("%e %a %x\n", f.val, f.val,  f.bin);
  int log_2 = ((f.bin >> 23) & 255) - 127;  //exponent
  f.bin &= 0x7FFFFF;                        //mantissa (aka significand)

  std::cout << "exp " << log_2 << " mant in binary "  << std::hex << f.bin
                << " mant as float " <<  std::dec << (f.bin|0x800000)*::pow(2.,-23)
                 << std::endl;
}
```

Examples of single precision fp

3f80 0000 (0x1p+0) = 1

c000 0000 (-0x1p+1) = −2

7f7f ffff (0x1.fffffep+127) ≈ 3.4028234×10^{38} (max single precision)

0000 0000 = 0

8000 0000 = −0

7f80 0000 (0x1.fffffep+127) = infinity

ff80 0000 () = -infinity

7f80 c000 = NaN

3eaa aaab (0x1.555556p-2) ≈ 1/3

Exercise:
Back to fcos.cpp
change cout in printf("%a\n")

std::numeric_limits<float> includes most of the info

Floating behaviour…

```
float a = 100.f+3.f/7.f;
 float b = 4.f/7.f;

  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;
```

Wha's the value of t?
How big is "t" (w.r.t. "s")?

```
cat precision.cpp
make precision
./precision_O2
```

```
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
```

Floating behaviour…

```
void f2(float x) {
  float y = std::sqrt(1.f/x/x);
  float z = 1.f/std::sqrt(x*x);
  std::cout << std::boolalpha;
  std::cout << z << " " << y << std::endl;
  std::cout << (z==y) << " " << (z<y) << " " << (z>y) << std::endl;
  printf("%e %e %e\n", x,y,z);
  printf("%a %a %a\n\n", x,y,z);
}
void f3(float x1, float x2) {
  float y = x1*x1 - x2*x2;
  float z = (x1+x2)*(x1-x2);
  std::cout << std::boolalpha;
  std::cout << x1 << " " << x2 << std::endl;
  std::cout << z << " " << y << std::endl;
  std::cout << (z==y) << " " << (z<y) << " " << (z>y) << std::endl;
  printf("%e %e %e %e\n", x1,x2,y,z);
  printf("%a %a %a %a\n\n", x1,x2,y,z);
}
```

make foperations
./foperations_O2

Floating Point Math

- Floating point numbers are NOT real numbers
  - They exist in a finite number (~2^{32} for single prec)
  - 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)
  - Results of Operations are rounded
    - Standard conformance requires half-ULP precision.
      - See next slide
    - $x + \varepsilon - x \neq \varepsilon$ (can be easily 0 or ∞)
  - Their algebra is not associative
    - $(a+b) + c \neq a+(b+c)$
    - $a/b \neq a*(1/b)$
    - $(a+b)*(a-b) \neq a^2 - b^2$

Rounding Algorithms

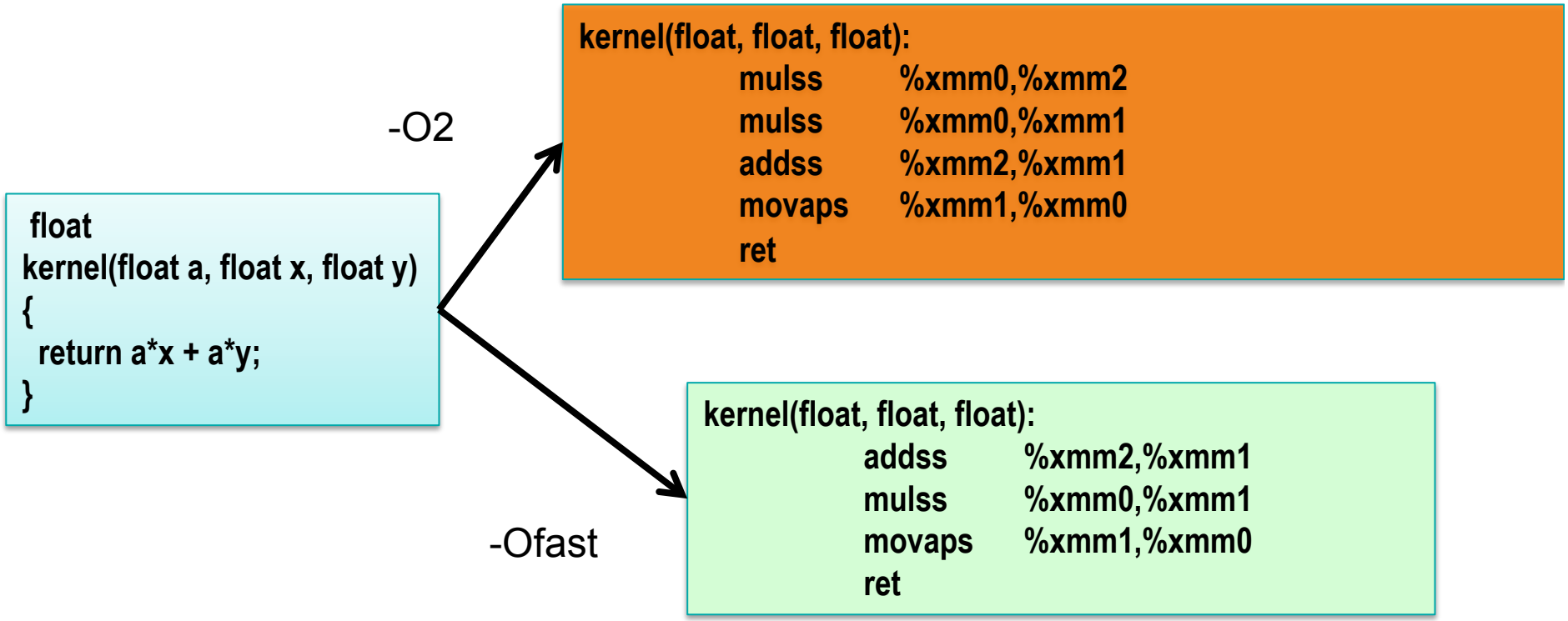
- The standard defines five rounding algorithms.
  - The first two round to a nearest value; the others are called directed roundings:
- Roundings to nearest
  - **Round to nearest, ties to even – rounds to the nearest value;** if the number falls midway it is rounded to the nearest value with an even (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 +∞ – directed rounding towards positive infinity (also known as rounding up or ceiling).
  - Round toward −∞ – directed rounding towards negative infinity (also known as rounding down or floor).

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
    - Caveat: the compiler improves continuously: still it does not optimize yet all kind of expressions

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)



Exercise: make assocMath; compare assembler for O2 and Ofast

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 mantissa
  - 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

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)

```
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 (sum.cpp)

```
//  Sum a list of pseudo-random numbers using various methods.
//  The array of numbers to sum is generated in two different way:
//     NOTNASTY: as they come: between -max and max
//     "NASTY":  large numbers for even values of the subscripts, small for odd values,
//        alternating positive and  negative i.e.
//        0: large positive, 1: small positive, 2: large negative,3: small negative, etc
// The methods used are
//   1.  simply sum the values in their array order.
//   2.  a highly accurate summation scheme.
//   3.  same as #2 except that the code is modified to allow vectorization to take place
//   4.  same as #1 except that the loop is split into 4 subloops; basically a reduction.
//        However, this scheme interacts with how the numbers are created (with the "nasty" option).
//    5.  same as #1 except that the loop is split into 5 subloops; basically a reduction.
//    6.  use an OpenMP reduction.
//    7.  sort the values by their value before adding them.
//    8.  sort the values by their magnitude before adding them.
//
//  The point of the exercise is to demonstrate the variation of results possible.
make clean; make sum ADDOPT="-fopenmp"; ./sum_02; ./sum_fast
make clean; make sum ADDOPT="-fopenmp -DNOTNASTY"; ./sum_02; ./sum_fast
```

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 exaclty
}
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
}
```

Almost Equal (source http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm

Comparing fp for equality is error prone

- result == expectedResult

is hardly satisfied because of rounding

- fabs(result - expectedResult) < ε
- fabs((result - expectedResult) / expectedResult) < ε

Not necessary better

A solution is to measure the “ulp gap” (next slide)

Almost equal

```
union fasi {
  int i;
  float f;
};
bool almostEqual(float a, float b, int maxUlps)
{
    // Make sure maxUlps is non-negative and small enough that the
    // default NAN won't compare as equal to anything.
    assert(maxUlps > 0 && maxUlps < 4 * 1024 * 1024);
    fasi fa; fa.f = a;
    // Make fa.i lexicographically ordered as a twos-complement int
    if (fa.i < 0) fa.i = 0x80000000 - fa.i;
    // Make fa.i lexicographically ordered as a twos-complement int
    fasi fb; b.f = b;
    if (fb.i < 0) fb.i = 0x80000000 - fb.i;
    int intDiff = std::abs(fa.i - fb.i)
    return (intDiff <= maxUlps);
}
```

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	DIV, SQRT	10-14	10-22	21-29	21-45
1.f/ , 1.f/sqrt	RCP, RSQRT	5		7	
=	MOV	1,3,…	1,3,…	1,4,….	1,4,…

Approximate reciprocal (sqrt, div)

The major contribution of game industry to SE is the discovery of the “magic” fast 1/√x algorithm

```
float InvSqrt(float x){
  union {float f;int i;} tmp;
  tmp.f = x;
  tmp.i = 0x5f3759df - (tmp.i >> 1);
  float y = tmp.f; // approximate
  return y * (1.5f - 0.5f * x * y * y); // better
}
```

Real x86_64 code for 1/√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 s	Gnu libm d	Cephes scalar s	Cephes scalar d	Cephes autovect s	Cephes autovect d	Cephes handvect s	Approx (16bits)	Intel svml s	Intel svml d	Amd libm s	Amd libm d
sin,cos	55	100	30		11		20		12	30	25	45
large x	***>500***		50		30							
sincos	70		40		15		22				50	
atan2	50	100	30		13				17	52	67	87
exp	***650***	65	42	55	10		27		12	26	16	36
					23							
log	50	105	37	42	11		24	12	12	30	27	59
					28							

SET_RESTORE_ROUND_NOEXF (FE_TONEAREST);

How to speed up math

- Avoid or factorize division and sqrt
  - if possible compile with “–Ofast –mrecip”
- 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 OptimizeIt.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

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

```
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;
}
```

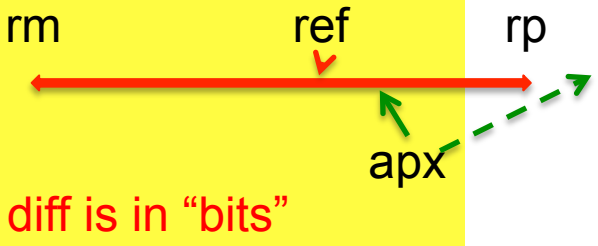
2nd order polynomial

Verify accuracy of approximation

```
float ref = ms(rl,m2,p2);
float rp = ms(rl*1.001,m2,p2);   // 0.1% positive
float rm = ms(rl*0.999,m2,p2);  // 0.1% negative
float apx = msf(rl,m2,p2);   // fast approximation

// look if approximation inside uncertainty-interval
int dd = std::min(abs(diff(rm,ref)),abs(diff(rp,ref)));
dd -= abs(diff(apx,ref)); // negative if apx-ref is larger than the uncer-interval
dm = std::min(dm,dd);

da = std::max(da,abs(diff(apx,ref)));  // maximum "error" by approx
di = std::max(di,abs(diff(rp,ref)));
di = std::max(di,abs(diff(rm,ref)));    // maximum uncertantly
// ditto for minimum
```



- 0.1% accuracy corresponds to a difference of 13-14 bits
- Maximum error of the approximation is ~12 bits
- "dm" always positive

One more exercise

```
float stripErrorSquared(const float uProj) {
    const Float P1=-0.339;
    const Float P2=0.90;
    const Float P3=0.279;
    const Float uerr = P1*uProj*std::exp(-uProj*P2)+P3;
    return uerr*uerr;
 }
```

What is the minimum precision we can afford on exp?
Assume error on parameters is 0.2 on the last digit
What about possible correlation? (factorize P1 out?)

SIMD: LOOP VECTORIZATION

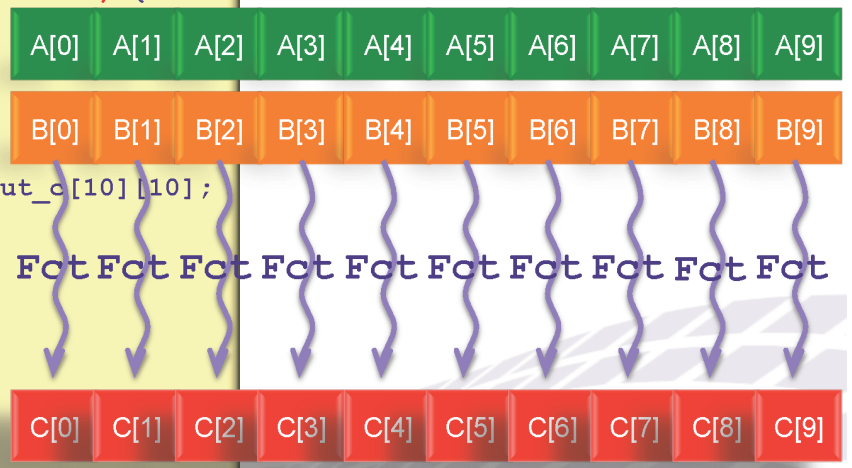
What is Stream Computing?

CAPS

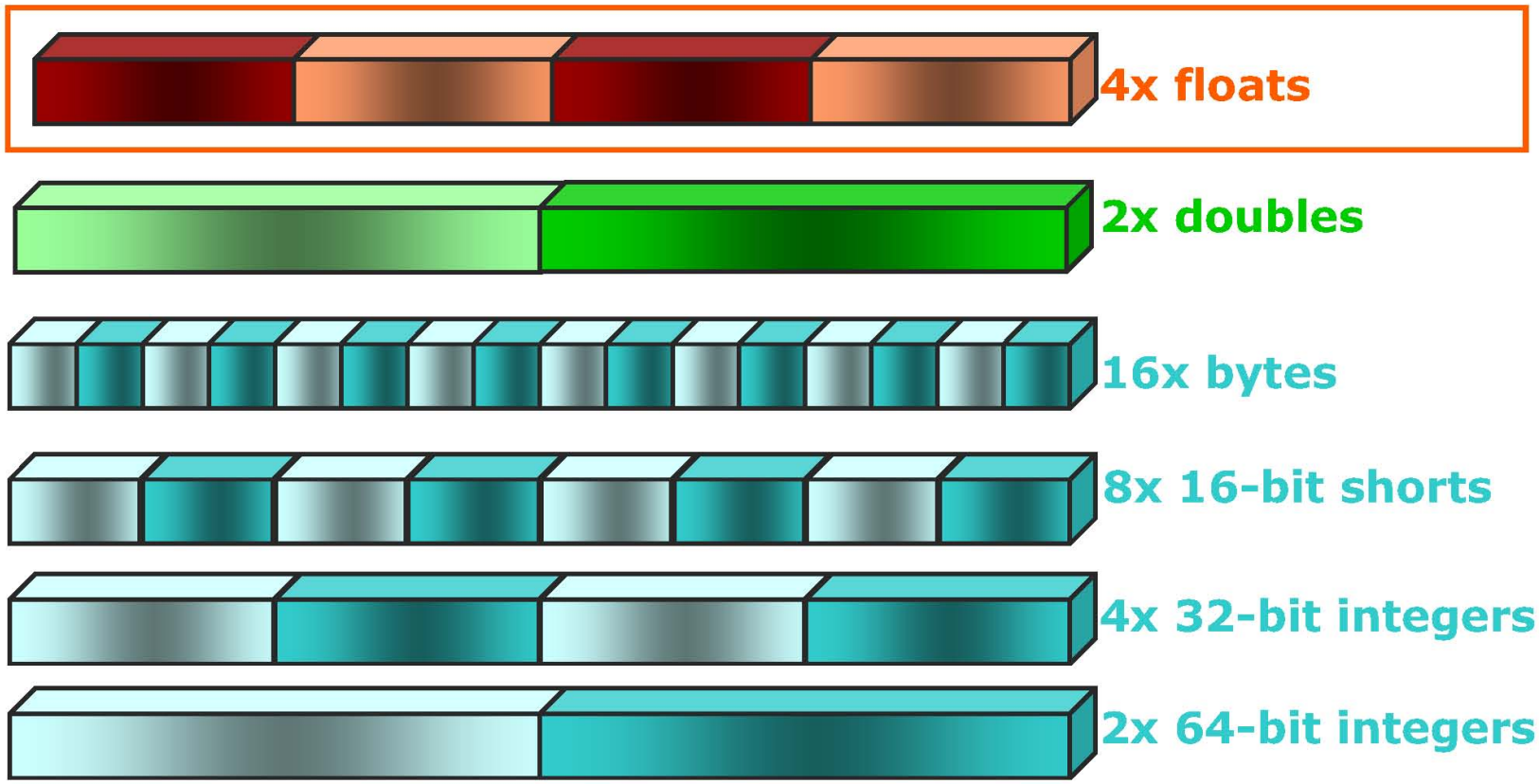
- A similar computation is performed on a collection of data (*stream*)
  - There is no data dependence between the computation on different stream elements
- Stream programming is well suited to GPU *and vector-cpu!*

```
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



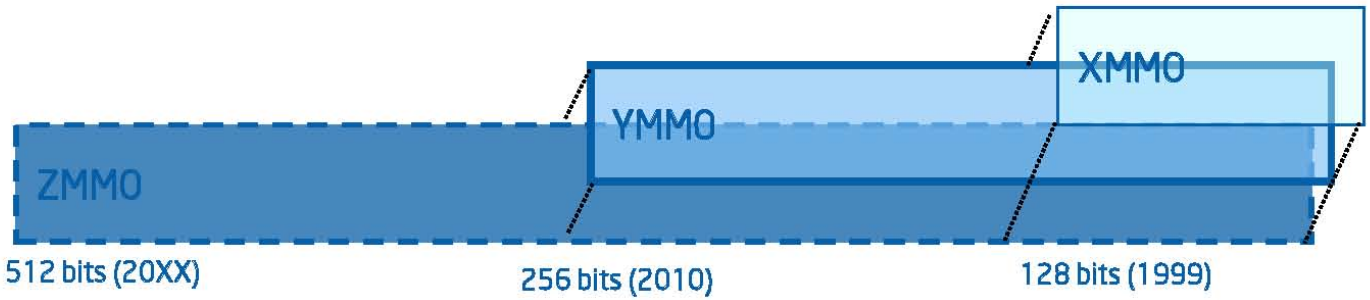
SSE Data types



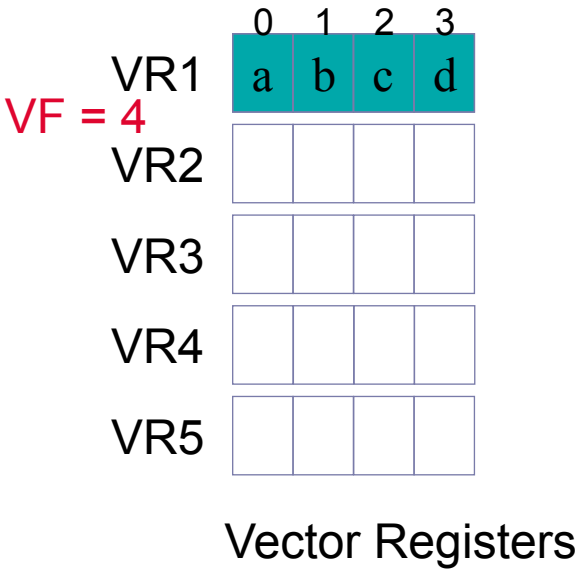
Sandy Bridge (2010): Intel® AVX

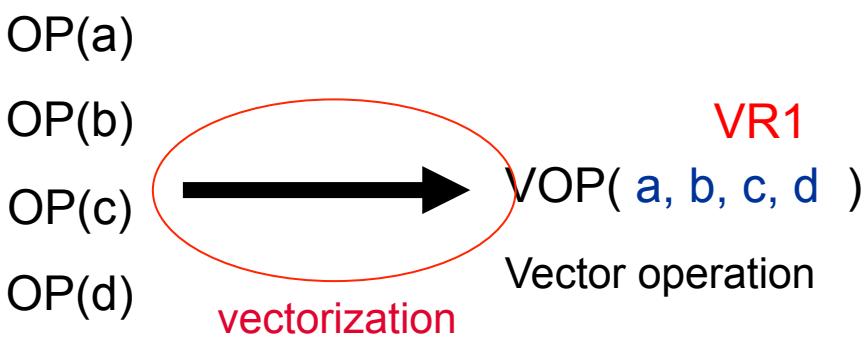
A 256-bit vector extension to SSE

- 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



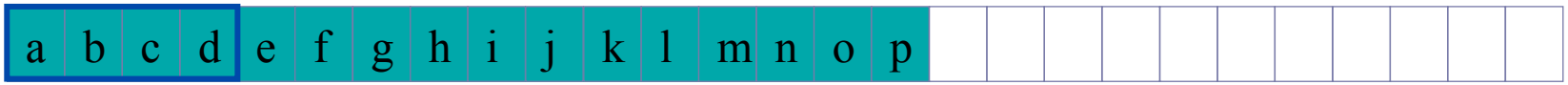
What is vectorization





- Data elements packed into vectors
- Vector length → 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; i+=VF){
   a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
}
```

- 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 based vectorization
- No dependences between iterations

SSE overview

- SSE = Streaming SIMD Extension
- SIMD – Single Instruction Multiple Data.
- One instruction to do the same operation on 4 packed elements simultaneously.

6 instructions / element

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

7 instructions / 4 elements

1.75 instructions / element

Scalar loop :

```
L1:
   movss   xmm0, [rdx+r13*4]
   mulss   xmm0, [r8+r13*4]
   movss   [rcx+r13*4], xmm0
   add     r13, 1
   cmp     r13, r9
   jl      L1
```

Vector loop :

```
L1:
   movups  xmm1, [rdx+r9*4]
   movups  xmm0, [r8+r9*4]
   mulps   xmm1, xmm0
   movaps  [rcx+r9*4], xmm1
   add     r9, 4
   cmp     r9, rax
   jl      L1
```

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
```

Reduction

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

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);
```

Requires *relaxed* float-math rules

Advanced exercise: vectorize Kahan summation in sum.cpp

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;
}
```

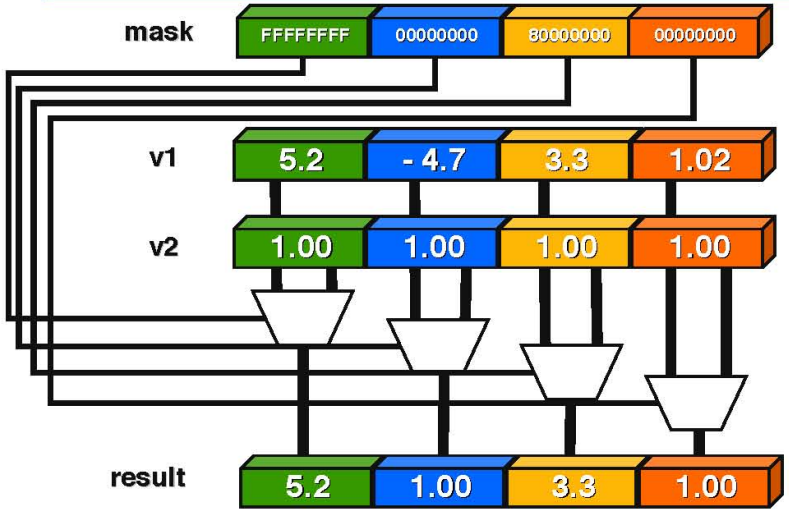
```
// pseudo code
void foo() {
   for (int i=0; i!=N; i++) {
      //evaluate condition
      int t = b[i]>a[i] ? ~0 : 0;
      // compute both branches
      float x= a[i]*b[i];
      float y=a[i];
      // mask and "blend"
      c[i] = t&x | ~t&y;
  }
}
```

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 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);
}
```

Exercise

- Open SimpleVectorization.cc
- make SimpleVectorization_vect.o
  - Analyze the compiler report
    - Correlate with code and generated instruction
    - Is vectorizing everything?
    - (if too complex, comment-out all but one function)
    - Add ADDOPT="-ftree-vectorizer-verbose=2”
  - Compare with
    - make SimpleVectorization_defvect.o

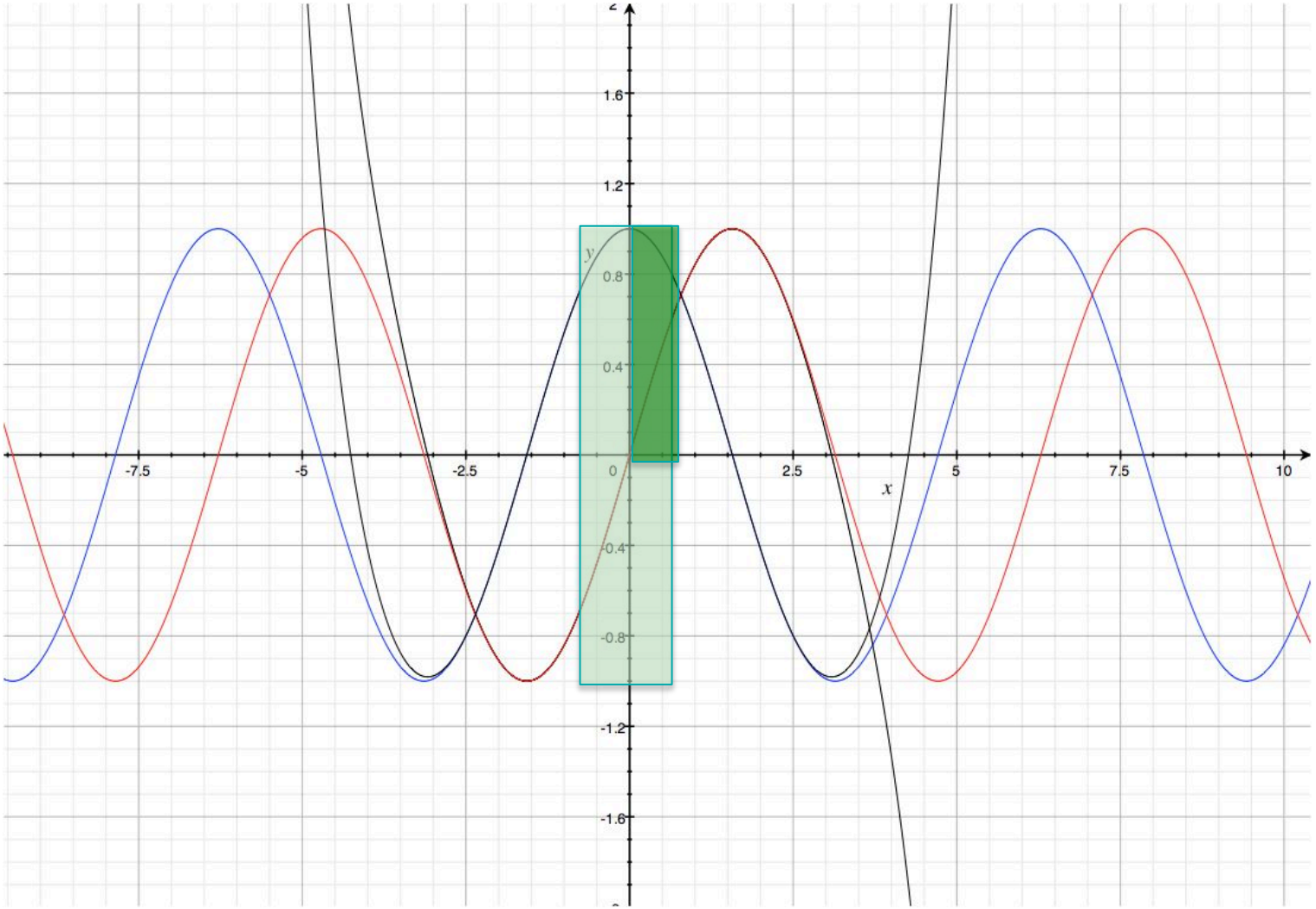
objdump -S -r -C --no-show-raw-insn –w xyz.o | less

conditional expression caveat

```
// The loops in foo are not vectorized. Those in bar are
float a1[1024], a2[1024],  b[1024], c[1024], d[1024], e[1024]; bool k[1024];
void foo () {
 for (int i = 0; i < 1024; i++) if(b[i] < c[i]) a1[i]=0;
 for (int i = 0; i < 1024; i++) a2[i] = b[i] < c[i] ? d[i] : e[i];
 for (int i = 0; i < 1024; i++) k[i] = b[i] < c[i]  && d[i] < e[i];
}
void bar (void){
for (int i = 0; i < 1024; i++)
   a1[i] = b[i] < c[i]  ? 0 : a1[i];
 for (int i = 0; i < 1024; i++) {
     float d_ = d[i], e_ = e[i];
     a2[i] = b[i] < c[i] ? d_ : e_;
   }
for (int i = 0; i < 1024; i++) k[i] = (b[i] < c[i]) &  (d[i] < e[i]);
}
```

gcc doesn't think it is ok to assign a1[i] or load d[i] resp. e[i] unconditionally.
(foo works for a1[256], e[256]; if b[i] > c[i] for i>255)

sin, cos



sin-cos: sequential vs vector optimization

Traditional sequential

```
int sign = 1;
if( x < 0 ) x = -x;

if( x > T24M1 ) return(0.0);
int j = FOPI * x;
float y = j;
if( j & 1 )  { j += 1;  y += 1.0; }
j &= 7;
if( j > 3) { j -=4; sign = -sign;}
if( j > 1 ) sign = -sign;

if( x > lossth ) x = x - y * PIO4F;
else x = ((x - y * DP1) - y * DP2) - y * DP3;
if( (j==1) || (j==2) ) cos = poly1(x);
else cos = poly2(x);
if(sign < 0) cos = -cos;
return cos;
```

Vector kernel

```
float x = fabs(xx);
// x =  (x > T24M1) ?   T24M1 : x;
int j = FOPI * x;
j = (j+1) & (~1);
float y = j;
int signS = (j&4);
j-=2;
int signC = (j&4);
int poly = (j&2);

x = ((x - y * DP1) - y * DP2) - y * DP3;
cos = poly1(x);
sin = poly2(x);
if( poly!=0 ) { swap(cos,sin);}
if(signC == 0) cos = -cos;
if(signS != 0) sin = -sin;
if (xx<0)  sin = -sin;
```

atan2: sequential vs vector optimization

Traditional sequential

```
code = 0;
if( x < 0.0 ) code = 2;
if( y < 0.0 ) code |= 1;
if( x == 0.0 ) {
  if( code & 1 ) return( -PIO2F );
  if( y == 0.0 ) return( 0.0 );
  return( PIO2F );
}
if( y == 0.0 ) {
  if( code & 2 ) return( PIF );
  return( 0.0 );
}
switch( code ) {
 default:
 case 0:
 case 1: w = 0.0; break;
 case 2: w = PIF; break;
 case 3: w = -PIF; break;
}
return w + atanf( y/x ); // more if,div + a poly
```

Vector kernel

```
float xx = fabs(x);
float yy = fabs(y);
float tmp =0.0f;
if (yy>xx) {   // pi/4
  tmp = yy; yy=xx; xx=tmp;
}
float t=yy/xx;
float z=t;
if( t > 0.41421356237730950f)   // pi/8
    z = (t-1.0f)/(t+1.0f); // always computed
// 2 divisions will cost more than the poly
float ret = poly(z);

if (y==0) ret=0.0f;
if( t > 0.41421356237730950f ) ret += PIO4F;
if (tmp!=0) ret = PIO2F - ret;
if (x<0) ret = PIF - ret;
if (y<0) ret = -ret;

return ret;
```

cos

```
float fast_cosf( float x ) {
  float y, z;
  /* make argument positive */
  int sign = 1;
  if( x < 0 ) x = -x;
  /* integer part of x/PIO4 */
  int j = FOPI * x;
  y = j;
  /* integer and fractional
      part modulo one octant */
  if( j & 1 ) {
      /* map zeros to origin */
      j += 1; y += 1.0;
   }
  j &= 7;
  if( j > 3) { j -=4; sign = -sign; }
  if( j > 1 ) sign = -sign;
```

```
  /* Extended precision modular arithmetic */
     x = ((x - y * DP1) - y * DP2) - y * DP3;

  z = x * x;
  if( (j==1) || (j==2) ) {  // vertical quadrants
      y = (((-1.9515295891E-4f * z
              + 8.3321608736E-3f) * z
             - 1.6666654611E-1f) * z * x)
        + x;
    } else {// horizontal quadrants
      y = ((  2.443315711809948E-005f * z
               - 1.388731625493765E-003f) * z
            + 4.166664568298827E-002f) * z * z
      - 0.5 * z + 1.0;
    }
  if(sign < 0) y = -y;
  return y;
}
```

sin

```
ftoat fast_sinf( float x ) {
  float y, z;
  int sign = 1;
  if( xx < 0 ) {
    sign = -1; x = -x;
  }
 /* integer part of x/(PI/4) */
 j = FOPI * x;
y = j;
 /* map zeros to origin */
  if( j & 1 ) {
    j += 1; y += 1.0;
  }
 /* octant modulo 360 degrees */
 j &= 7; /* reflect in x axis */
 if( j > 3) { sign = -sign; j -= 4; }
```

```
  /* Extended precision modular arithmetic */
   x = ((x - y * DP1) - y * DP2) - y * DP3;

z = x * x;
if( (j==1) || (j==2) ) {
   y = ((  2.443315711809948E-005f * z
          - 1.388731625493765E-003f) * z
        + 4.166664568298827E-002f) * z * z
    - 0.5 * z + 1.0;
  } else {
  y = (((-1.9515295891E-4f * z
         + 8.3321608736E-3f) * z
        - 1.6666654611E-1f) * z * x)
      + x;
  }
if(sign < 0) y = -y;
  return y;
}
```

sincos (vectorizable)

```
// reduce to -45deg<x< 45deg, return also quadrant
inline float reduce2quadrant(float x, int & quad) {
  /* make argument positive */
    x = fabs(x);
    quad = FOPI * x; /* integer part of x/PIO4 */
    quad = (quad+1) & (~1);
    float y = quad;
    // Extended precision modular arithmetic
   return ((x - y * DP1) - y * DP2) - y * DP3;
}
// only for  -45deg < x < 45deg
inline void sincosf0( float x, float & s, float &c ) {
  float z = x * x;
  s = x + (((-1.9515295891E-4f * z
               + 8.3321608736E-3f) * z
              - 1.6666654611E-1f) * z * x);
   c = ((  2.443315711809948E-005f * z
            - 1.388731625493765E-003f) * z
           + 4.166664568298827E-002f) * z * z
     - 0.5f * z + 1.0f;
  }
```

```
inline void sincosf(
  float xx, float & s, float &c ) {
  float ls,lc;
  int j=0;
  float x = reduce2quadrant(xx,j);
  int signS = (j&4);
  j-=2;
  int signC = (j&4);

  int poly = j&2;
  sincosf0(x,ls,lc);
  if( poly==0 ) swap(ls,lc);

  if(signC == 0) lc = -lc;
  if(signS != 0) ls = -ls;
  if (xx<0)  ls = -ls;
  c=lc;
  s=ls;
}
```

Difficult (impossible?) to vectorize

```
// recursion
void gen(float const * rd, int& j) {
   for (int i=0; i!=N; i++) {
      do{
        xx= 2*rd[j++]-1;
        yy =2*rd[j++]-1;
        r2 = xx*xx+yy*yy;
      } while (r2>1||r2==0);
      m= sqrt(-2 * log(r2) /r2);
      x[i]=m*xx; y[i]=m*yy;
  }
}
```

```
// Array of structures
struct P {float x,y,z;};
void foo(P * p, int N, float phi) {
   for (int i=0; i!=N; i++) {
      float xx=p[i].x, yy=p[i].y;
      p[i].x =  sin(phi)*xx+cos(phi)*yy;
      p[i].y = -cos(phi)*xx+sin(phi)*yy;
  }
}
```

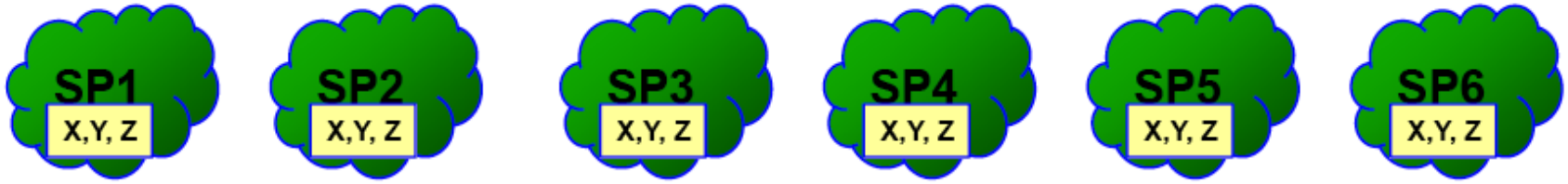
```
// lookup
const float radius[12] = { … …};
void bar() {
   for (int i=0; i!=N; i++) {
      r = radius[layer[i]];
      x=cos(phi[i])/r;
      ++h[int(x/x0)];
  }
}
```

Possible solutions

- Recursion
  - Change algorithm!
  - (splitting the loop may work in some cases)
- Array of Structures (AOS)
  - Use Structure of Arrays (SOA)
  - (compiler can also “stride-vectorize”)
- Lookup
  - split loop, vectorize “heavy-computational” part
  - (one may have to change algorithm)

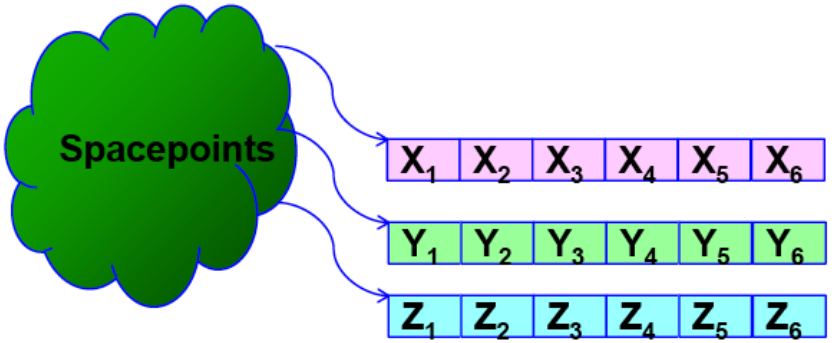
Data Organization: AoS vs SoA

- 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)

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
- Explot auto-vectorization
  - Modified cephes library (or other open source)
    - Requires header-file + different function names
  - Look into vdt/

GCC idiosyncrasies (GCC 4.7)

- Induction variables (loop indices): local “int”
- Use local variables inside loops
  - Avoid to update memory locations
  - --param vect-max-version-for-alias-checks=NNN
  - -ftree-loop-if-convert-stores
- Often to trigger vectorization “loop unrolling” needs to be inhibited
  - --param max-completely-peel-times=1
- Sometime better to inhibit other optimization
  - -no-tree-pre

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

Exercises

- Vectorize quadratic equation solution
  - Find fastest algorithm with “good accuracy”
- Take atan2 (or asin)
  - Compare speed for limited range (<20degree) w.r.t. full range
- Use approx_exp (or log)
  - Compare Estrin vs Horner (code Estrin for log!)
  - Measure the effects of “if”s (approx_ vs unsafe_)
  - (all this is more on ILP than vectorization)

One more exercise (chi2.cc)

```
typedef float Scalar;
// typedef double Scalar;
constexpr int SIZE=10;
Scalar overab, offset, alphaS;
void compChi2(Scalar const * ampl, Scalar const * err2, Scalar t,
              Scalar sumAA, Scalar& chi2, Scalar& amp) {
  Scalar sumAf = 0;
  Scalar sumff = 0;
  constexpr Scalar eps = Scalar(1e-6);
  constexpr Scalar denom =  Scalar(1)/Scalar(SIZE);

  for(unsigned int it = 0; it < SIZE; it++){
    Scalar offset = (Scalar(it) - t)*overabS;
    Scalar term1 = Scalar(1) + offset;
    if(term1>eps){
      Scalar f = std::exp( alphaS*(std::log(term1) - offset) );
      sumAf += ampl[it]*(f*err2[it]);
      sumff += f*(f*err2[it]);
    }
  }
  chi2 = sumAA;
    amp = 0;
    if( sumff > 0 ){
      amp = sumAf/sumff;
      chi2 = sumAA - sumAf*amp;
    }
    chi2 *=denom;
}
```

ADVANCED TOPICS

SLP (Superword Level Parallelism)

- SLP (vectorization of straight line code) consists in transforming multiple identical instructions in vector code
- SLP is in general less efficient than loop vectorization
  - But applies also in otherwise “scalar code”

```
struct LorentzVector {
  LorentzVector(T x=0, T  y=0, T  z=0, T  t=0) :
   theX(x),theY(y),theZ(z),theT(t){}
  T theX;
  T theY;
  T theZ;
  T theT;
}  __attribute__ ((aligned(16)));
```

Exercise:
make FourVect and inspect code
benchmark SLP vs loop

```
inline LorentzVector
operator+(LorentzVector const & a, LorentzVector const & b) {
  return LorentzVector(a.theX+b.theX,a.theY+b.theY,a.theZ+b.theZ,a.theT
+b.theT);
}
```

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*

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

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

Floating Point Operations, cache access

Branches, function calls,

Memory access, virtual calls, malloc

malloc, I/O, IPC

Services, database,

User time

System time

Wait

One More example

- In CMS the Vavilov distribution is used to compute the probability of a cluster in a Silicon Detector to come from a m.i.p.
  - It is then encoded in an 8-bit quality word
- Precision tuned-down while verifying that the final result (the 8-bits!) do not change
- Speed up of a factor 3…

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 $y_n^{(i)}$, $1 \le i \le 5$, and define

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

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, 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.

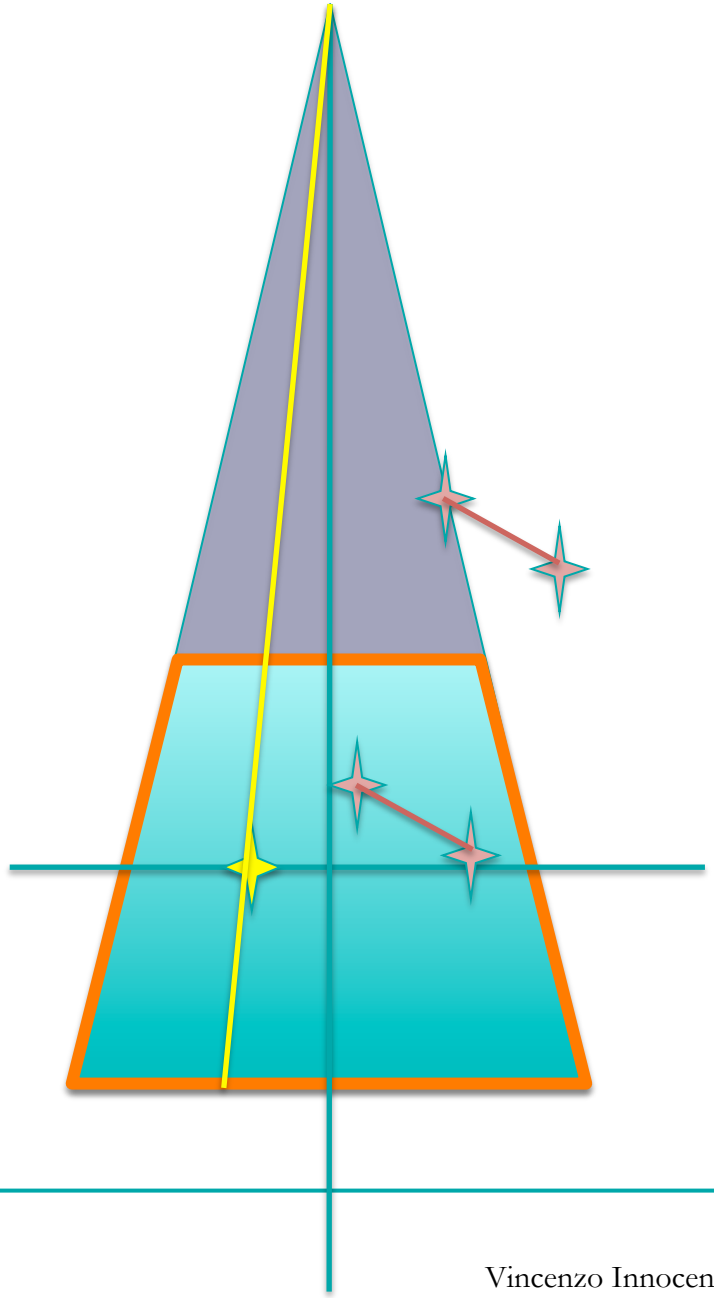
Suppose now that we were to accept the solution of order 5 at x_n. We wish to compute a suitable step length, $\bar{h}_4$, to be used in integrating from x_n to x_{n+1} using a 5(4) formula. A typical step-choosing strategy would compute $\bar{h}_4$ as

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

Here ϵ 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

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

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



Summary of Exercises

- Count the number of “floats” between 100 and 101
- Use Kahan summation, measure speed
  - Compare with double precision
  - Try to vectorize
- Improve performance of code in slide 31
- 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
- Measure the impact of “conditions”
- take code from either vdt::fast_sincosf, vdt::fast::atan2f or approx_logf/ approx_expf
- use it in a restricted range
- Compare SoA to AoS approach for instance in the rotation kernel