



## 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

cd hands-on

cd floatingpoint

c++ -O2 -Wall -march=native floats.cpp

./a.out

Try

<https://gcc.godbolt.org/>

---

# 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
  - INTEL doc and white papers (including CoIFax web site)
  - MetaLibm site

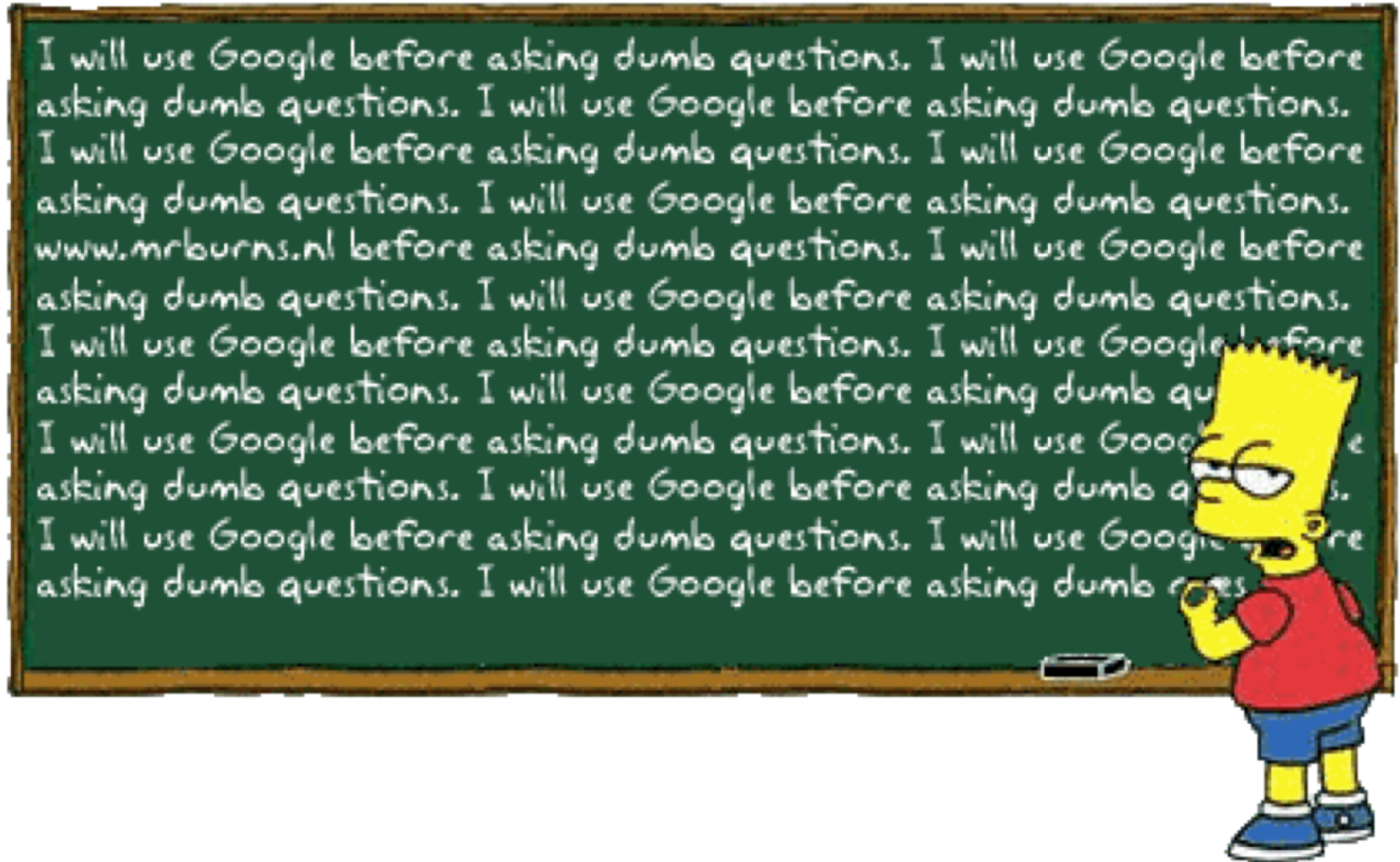


---

# Applicability

- It is very (very) different if you deal with
  - Video games
  - 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-kahan/>

# Don't be afraid to ask questions!

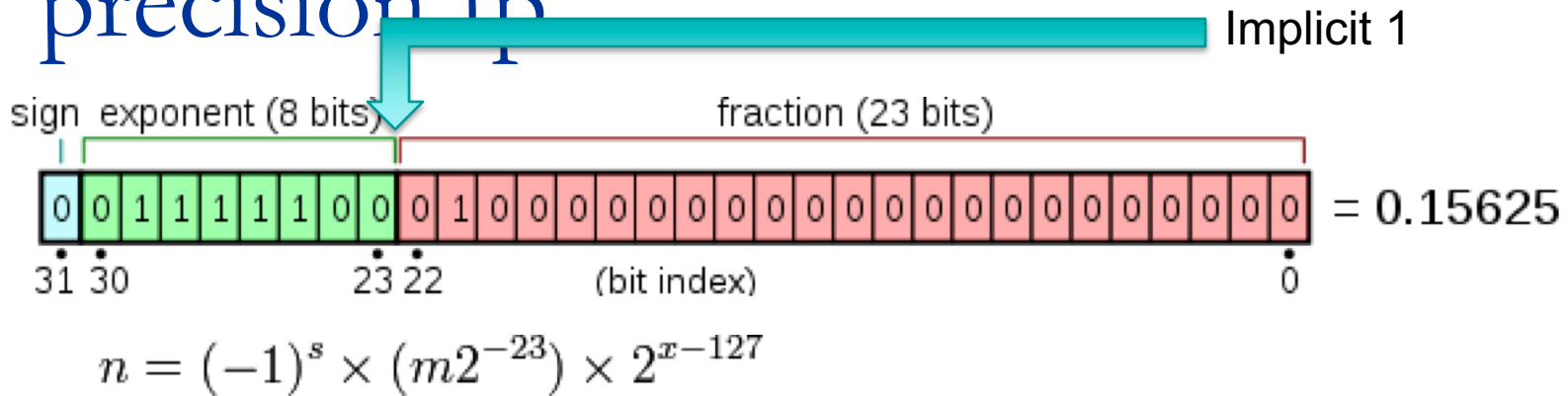


# Floating Point Representation

(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  $3.4 \times 10^{38}$
    - double:  $\sim 10^{-323}$  to  $\sim 10^{308}$
  - This has a cost...

# IEEE 754 representation of single precision fp



| 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>                        | $\pm$ infinity   | <u>NaN (quiet,signalling)</u> |   |

# Floating Point Types

| Type                   | Sign | Exponent | Significand field | Total bits | Exponent bias | Bits precision | Number of decimal digits |
|------------------------|------|----------|-------------------|------------|---------------|----------------|--------------------------|
| Half (IEEE 754-2008)   | 1    | 5        | 10                | 16         | 15            | 11             | ~3.3                     |
| Single                 | 1    | 8        | 23                | 32         | 127           | 24             | ~7.2                     |
| Double                 | 1    | 11       | 52                | 64         | 1023          | 53             | ~15.9                    |
| x86 extended precision | 1    | 15       | 64                | 80         | 16383         | 64             | ~19.2                    |
| Quad                   | 1    | 15       | 112               | 128        | 16383         | 113            | ~34.0                    |

- Half : `__fp16`, available for storage on recent ARM, INTEL and NVIDIA HW
- Single: `float`, storage and computation in C++ on any hardware
- Double: `double`, storage and computation in C++ on any hardware
- Ext-precision: `long double`, No storage! computation in C++ on x86 hardware
- Quad: `__float128`, storage and (software) computation in C++

# C++ interlude

```
#include<iostream>
#include<iomanip>
#include<cmath>
#include<limits>
// std style
template<typename T>
void print(T x) {
    std::cout << std::hexfloat << x << ' ' // exact binary representation
               << std::scientific << std::setprecision(8) << x << ' ' // 8 decimal digits
               << std::defaultfloat << x << std::endl; // 6 decimal digits for float
}

// integer representation
#include<cstring>
#include<bitset>
int f2i(float x) { int i; memcpy(&i,&x,sizeof(int)); return i;} // compiler will optimize
// use bitset to print in binary (bit by bit)
std::cout << std::bitset<32>(f2i(x)) << std::endl; ; // bit by bit
```

# 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/)

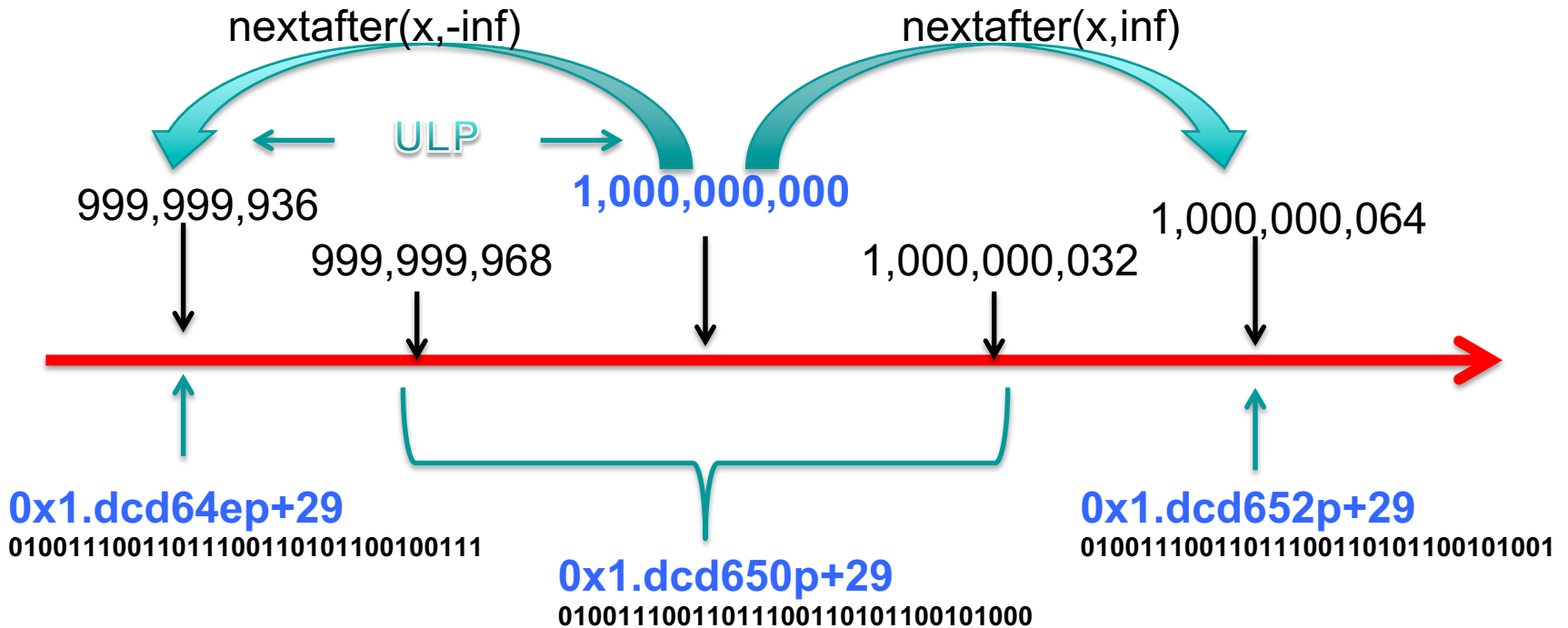
```
#include <iostream>    // std::cout
#include <limits>      // std::numeric_limits
int main () {
    std::cout << std::boolalpha;
    std::cout << "Minimum value: " << std::numeric_limits<float>::min() << "\n";
    std::cout << "Maximum value: " << std::numeric_limits<float>::max() << "\n";
    std::cout << "Is signed: " << std::numeric_limits<float>::is_signed << "\n";
    std::cout << "significand bits: " << std::numeric_limits<float>::digits << "\n";
    std::cout << "precision: " << std::numeric_limits<float>::epsilon() << "\n";
    std::cout << "has infinity: " << std::numeric_limits<float>::has_infinity << "\n";
    return 0;
}
```

# examples

|                |                 | sign | exponent | significant              |
|----------------|-----------------|------|----------|--------------------------|
| 0.00000000e+00 | 0x0p+0          | 0    | 00000000 | 000000000000000000000000 |
| 5.00000000e-01 | 0x1p-1          | 0    | 01111111 | 000000000000000000000000 |
| 1.00000000e+00 | 0x1p+0          | 0    | 01111111 | 000000000000000000000000 |
| 1.50000000e+00 | 0x1.8p+0        | 0    | 01111111 | 100000000000000000000000 |
| 2.00000000e+00 | 0x1p+1          | 0    | 10000000 | 000000000000000000000000 |
| 2.50000000e+00 | 0x1.4p+1        | 0    | 10000000 | 010000000000000000000000 |
| 3.00000000e+00 | 0x1.8p+1        | 0    | 10000000 | 100000000000000000000000 |
| 8.00000000e+00 | 0x1p+3          | 0    | 10000010 | 000000000000000000000000 |
| 3.14159274e+00 | 0x1.921fb6p+1   | 0    | 10000000 | 10010010000111111011011  |
| 1.00000001e-01 | 0x1.99999ap-4   | 0    | 01111011 | 10011001100110011001101  |
| 1.17549435e-38 | 0x1p-126        | 0    | 00000001 | 000000000000000000000000 |
| 3.40282347e+38 | 0x1.fffffep+127 | 0    | 11111110 | 111111111111111111111111 |
| nan            | nan             | 1    | 11111111 | 100000000000000000000000 |
| -inf           | -inf            | 1    | 11111111 | 000000000000000000000000 |
| 2.93873588e-39 | 0x1p-128        | 0    | 00000000 | 010000000000000000000000 |



# Precision, rounding



```
float x = 1000*1000*1000;
```

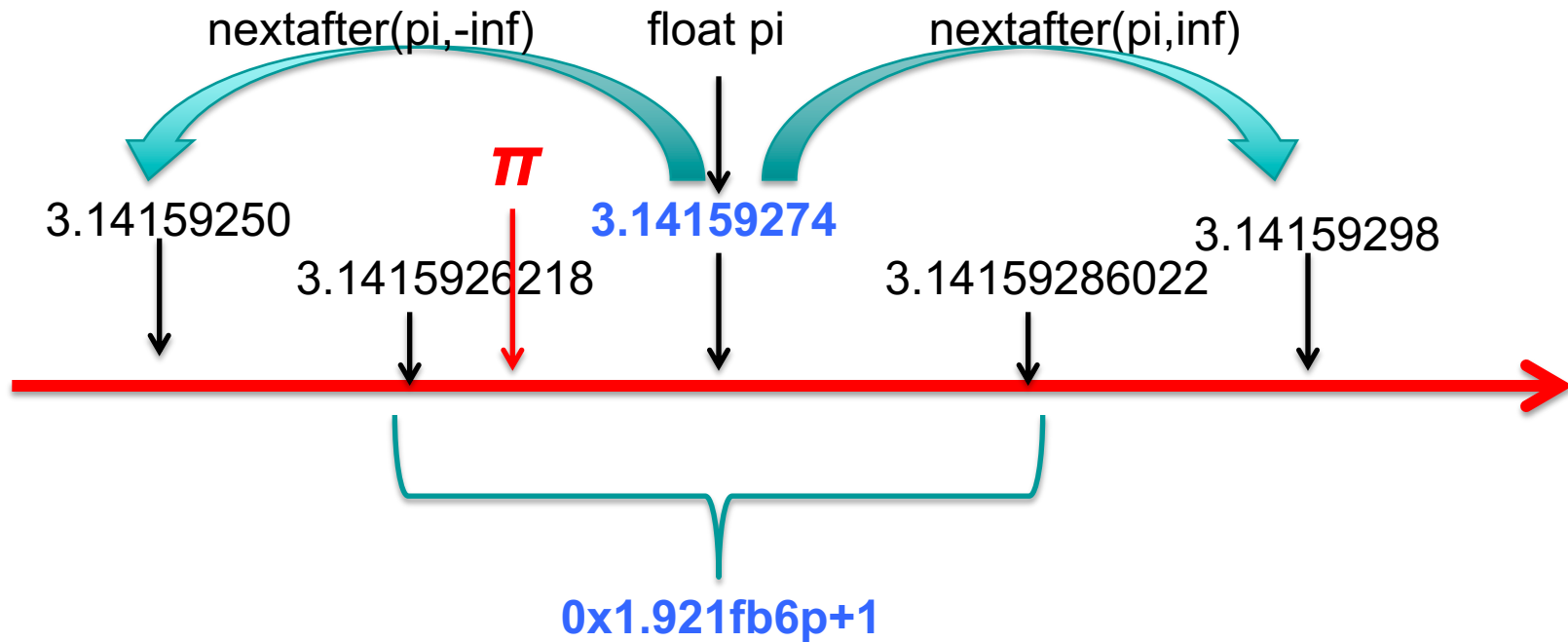
```
std::cout << std::scientific << std::setprecision(8)  
  << x << ' ' << x+32.f << ' ' << x+33.f << std::endl  
  << x+32.f-x << x+33.f-x << std::endl;
```

```
1.00000000e+09 1.00000000e+09 1.00000006e+09
```

```
0.00000000e+00 6.40000000e+01
```

[https://en.wikipedia.org/wiki/Unit\\_in\\_the\\_last\\_place](https://en.wikipedia.org/wiki/Unit_in_the_last_place)

# Precision, rounding



`float pi = M_PI; // 0x1.921fb54442d18p+1 3.1415926535897931e+00`

← Double precision!

# 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).

# Loss of precision

## Summing *many* values

```
float x1=0;
for (float y=1;y<=1000000; ++y) x1+=y;
print (x1); // 4.99941376e+11 4.9994138e+11 0x1.d19b5p+38
```

```
float x2=0;
for (float y=1000000;y>0; --y) x2+=y;
print (x2); // 4.99873677e+11 4.9987368e+11 0x1.d18b2cp+38
```

```
int N=16; float xx[N]; for (int i=0; i<N; ++i) xx[i]=0;
for (int i=0; i<N; ++i)
for (int y=i;y<=1000000; y+=N) xx[i]+=float(y);
float s=0; for (int i=0; i<N; ++i) s+=xx[i];
print (s); // 4.99999900e+11 4.999999e+11 0x1.d1a944p+38
```

```
float result = 0.5*1000000 *(1000000+1);
std::cout << std::scientific << (result-x1)/result << " "
          << (result-x2)/result << " " << (result-s)/result << std::endl;
// 1.18226832e-04 2.53624079e-04 1.17964680e-06
```

```
float x3=0; float tenth=0.1f;
for(int i=0; i<10000;++i) x3+=tenth;
print(x3); // 9.99902893e+02 999.90289 0x1.f3f392p+9 (google for: patriot failure)
```

# Loss of precision

Cancellations: Subtracting *too-close* values

For example, consider the quadratic equation:

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

with the two exact solutions:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Better solution

$$x_1 = \frac{-b - \operatorname{sgn}(b) \sqrt{b^2 - 4ac}}{2a},$$

$$x_2 = \frac{2c}{-b - \operatorname{sgn}(b) \sqrt{b^2 - 4ac}} = \frac{c}{ax_1}.$$

**Two sources of cancellation**



[https://en.wikipedia.org/wiki/Loss\\_of\\_significance](https://en.wikipedia.org/wiki/Loss_of_significance)

# Loss of precision

[To illustrate the instability of the standard quadratic formula *versus* this variant formula, consider a quadratic equation with roots 1.786737589984535 and  $1.149782767465722 \times 10^{-8}$ . To sixteen significant figures, roughly corresponding to double-precision accuracy on a computer, the monic quadratic equation with these roots may be written as:

$$x^2 - 1.786737601482363x + 2.054360090947453 \times 10^{-8} = 0$$

Using the standard quadratic formula and maintaining sixteen significant figures at each step, the standard quadratic formula yields

$$\begin{aligned}\sqrt{\Delta} &= 1.786737578486707 \\ x_1 &= (1.786737601482363 + 1.786737578486707)/2 = 1.786737589984535 \\ x_2 &= (1.786737601482363 - 1.786737578486707)/2 = 0.000000011497828\end{aligned}$$

Note how cancellation has resulted in  $x_2$  being computed to only eight significant digits of accuracy. The variant formula presented here, however, yields the following:

$$\begin{aligned}x_1 &= (1.786737601482363 + 1.786737578486707)/2 = 1.786737589984535 \\ x_2 &= 2.054360090947453 \times 10^{-8} / 1.786737589984535 = 1.149782767465722 \times 10^{-8}\end{aligned}$$

Note the retention of all significant digits for  $x_2$ .

# 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 default is “strict IEEE754 mode”
  - `-O2 -funsafe-math -ffast-math`
  - `-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`

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

-O2

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

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

-Ofast

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

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

Exercise: compare assembler for O2 and Ofast  
<http://goo.gl/5jwOVO>

# FMA (fused multiply add)

- $\text{fma}(a,b,c) = \text{round}(a*b+c)$ 
  - Opposed to  $\text{round}(\text{round}(a*b)+c)$
- Single instruction with 4 or 5 cycle latency
  - Opposed to 2 instructions with 5+3 cycle latency
- More precise (one rounding instead of two)
  - Results will differ!
- Introduce a “contraction” issue
  - No standard contraction rules (yet)
  - $a*b+c*d$  ,  $a+b*c+d$  ,  $x*=a;x+=b$  , etc

---

# Controlling fma (contraction)

- gcc (and clang) flags
  - `-ffp-contract=off` (default is “fast”)
  - <https://stackoverflow.com/questions/43352510/difference-in-gcc-ffp-contract-options>
- cuda (nvcc) flags
  - `-fmad=false` (default is “true”)
  - <https://docs.nvidia.com/cuda/floating-point/index.html>

# fma library function

- `std::fma`
  - Implemented in software (libm)
    - Used if fma instruction not supported: SLOW
  - Compilers will use fma instruction if exists
    - Not affected by re-association flags
- To fully control fma and be fully reproducible
  - use `std::fma`
  - disable “contraction”

# 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

# 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.

## Defensive Computing

- Verify pre/post conditions and algorithm invariants
- Strict IEEE754 conformance
- In case of invalid input notify
  - Exception
  - Error code
  - Nan, inf..
- Can be implemented as a wrapper of a HPC algo

## High Performance

- Garbage in, garbage out
- Assume finite math
- Invalid input
  - Undefined behavior
  - Algorithm processes it at no additional cost
- User responsibility to guarantee correctness

In <https://gcc.gnu.org/z/OmNMxx> change -O3 in -Ofast

---

# **PRECISION, ACCURACY, SPEED**



---

# Definitions (mine)

## ■ Precision

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

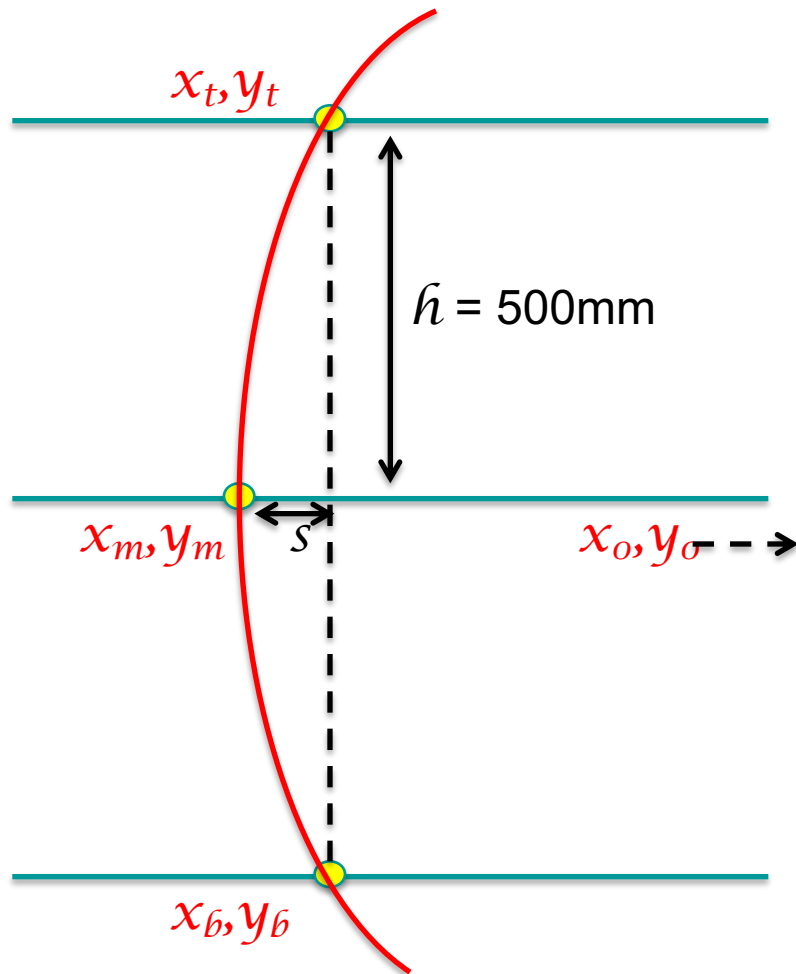
## ■ Accuracy

- ❑ 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 known error on the truth

Example: detector of size 1 m and resolution 10 micron  
 (required accuracy:  $\sim 1$ micron?)



three measurement planes

Is single precision enough to represent

- geometric elements?
- Measurements?
- computed trajectory?

Is enough to perform computation?

Trajectory: circle with radius  $r$   
 $s$  is the sagita,  $\hat{h}$  the half-chord

$$r = (s^2 + \hat{h}^2) / 2s \sim \hat{h}^2 / 2s$$

$$\hat{h}^2 = 2sr - s^2$$

$$x_o = r + x_m$$

# Circle equation and solutions

$$(x-x_0)^2 + (y-y_0)^2 = r^2$$

Compute  $x_t$  (x for  $y=h$ )

$$x = x_0 - (r^2 - h^2)^{1/2} \sim 0 \text{ (zero)}$$

Alternative formulation equivalent to change of coordinate centered at  $x_m, y_m$

$$(x-x_m)^2 + (y-y_m)^2 - 2ar(x-x_m) - 2br(y-y_m) = 0$$

In our case  $a=1, b=0$

$$(x-x_m)^2 - 2(x-x_m)r + h^2 = 0$$

$$x = x_m + h^2 / (r + (r^2 - h^2)^{1/2}) \sim x_m + h^2 / 2r$$

A correct choice of algorithm can allow to use a reduced precision to obtain the same accuracy

- Faster
- Smaller memory footprint

---

# **PRECISION, ACCURACY, SPEED**

# Cost of operations (in cpu cycles)

| op                 | instruction      | sse s        | sse d        | avx s        | avx d (FMA s&d)                       |
|--------------------|------------------|--------------|--------------|--------------|---------------------------------------|
| +,-                | ADD,SUB          | 3            | 3            | 3            | 3 4                                   |
| ==<br>< >          | COMISS CMP..     | 2,3          | 2,3          | 2,3          | 2,3                                   |
| f=d<br>d=f         | CVT..            | 3            | 3            | 4            | 4                                     |
| ,& ,^              | AND,OR           | 1            | 1            | 1            | 1                                     |
| *                  | MUL              | 5            | 5            | 5            | 5 4                                   |
| <b>/,sqrt</b>      | <b>DIV, SQRT</b> | <b>10-14</b> | <b>10-22</b> | <b>21-29</b> | <b>21-45</b>                          |
| 1.f/ ,<br>1.f/sqrt | RCP, RSQRT       | 5            |              | 7            |                                       |
| =                  | MOV              | 1,3,....     | 1,3,....     | 1,4,.....    | 1,4,.... <b>→350 from main memory</b> |

# Approximate reciprocal (sqrt, div)

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

[https://en.wikipedia.org/wiki/Fast\\_inverse\\_square\\_root](https://en.wikipedia.org/wiki/Fast_inverse_square_root)

```
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<br>(~IEEE754 compliant) |        | VDT (Cephes)<br>(~1ULP precise) |        |
|------------------------|----------------------------------|--------|---------------------------------|--------|
|                        | single                           | double | single                          | double |
| sin,cos,tan<br>large x | 55<br><b>&gt;500</b>             | 100    | 30                              | 50     |
| sincos                 | 70                               |        | 40                              |        |
| atan2                  | 50                               | 100    | 30                              | 45     |
| exp                    | <b>650</b>                       | 65     | 42                              | 55     |
| log                    | 50                               | 105    | 37                              | 42     |

`SET_RESTORE_ROUND_NOEXF (FE_TONEAREST);`

# Precision vs Speed in “libm”

## ■ Log

- 24 bit precision needs 8<sup>th</sup> degree polynomial
- 16 bit precision needs 5<sup>th</sup> degree polynomial
- 7 bit precision needs 2<sup>th</sup> degree polynomial

|              | 2   | 3    | 4    | 5    | 6    | 7    | 8    |
|--------------|-----|------|------|------|------|------|------|
| $e^x$        | 6.8 | 10.8 | 15.1 | 19.8 | 24.6 | 29.6 | 34.7 |
| $\sin(x)$    | 7.8 | 12.7 | 16.1 | 21.6 | 25.5 | 31.3 | 35.7 |
| $\ln(1+x)$   | 8.2 | 11.1 | 14.0 | 16.8 | 19.6 | 22.3 | 25.0 |
| $\arctan(x)$ | 8.7 | 9.8  | 13.2 | 15.5 | 17.2 | 21.2 | 22.3 |
| $\tan(x)$    | 4.8 | 6.9  | 8.9  | 10.9 | 12.9 | 14.9 | 16.9 |
| $\arcsin(x)$ | 3.4 | 4.0  | 4.4  | 4.7  | 4.9  | 5.1  | 5.3  |
| $\sqrt{x}$   | 3.9 | 4.4  | 4.8  | 5.2  | 5.4  | 5.6  | 5.8  |

Number of significant bits of accuracy vs degree of minimax polynomial approximating the range [0..1].



# 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

---

# 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, verify speed gain

---

# Formula Translation: the solution?



Is the compiler able to do it for us?

---

---

# **SIMD: LOOP VECTORIZATION**

---

# Test environment

```
cd hands-on/vectorization
```

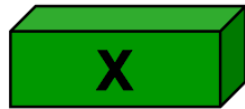
```
c++ -O2 pi.cpp -fopt-info-vec -march=native;  
time ./a.out
```

```
c++ -Ofast pi.cpp -fopt-info-vec -march=native;  
time ./a.out
```

# Single Instruction Multiple Data

- **Scalar processing**

- traditional mode
- one operation produces one result



+

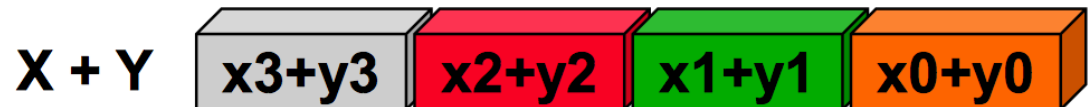


- **SIMD processing (Intel)**

- with SSE / SSE2
- one operation produces multiple results



+

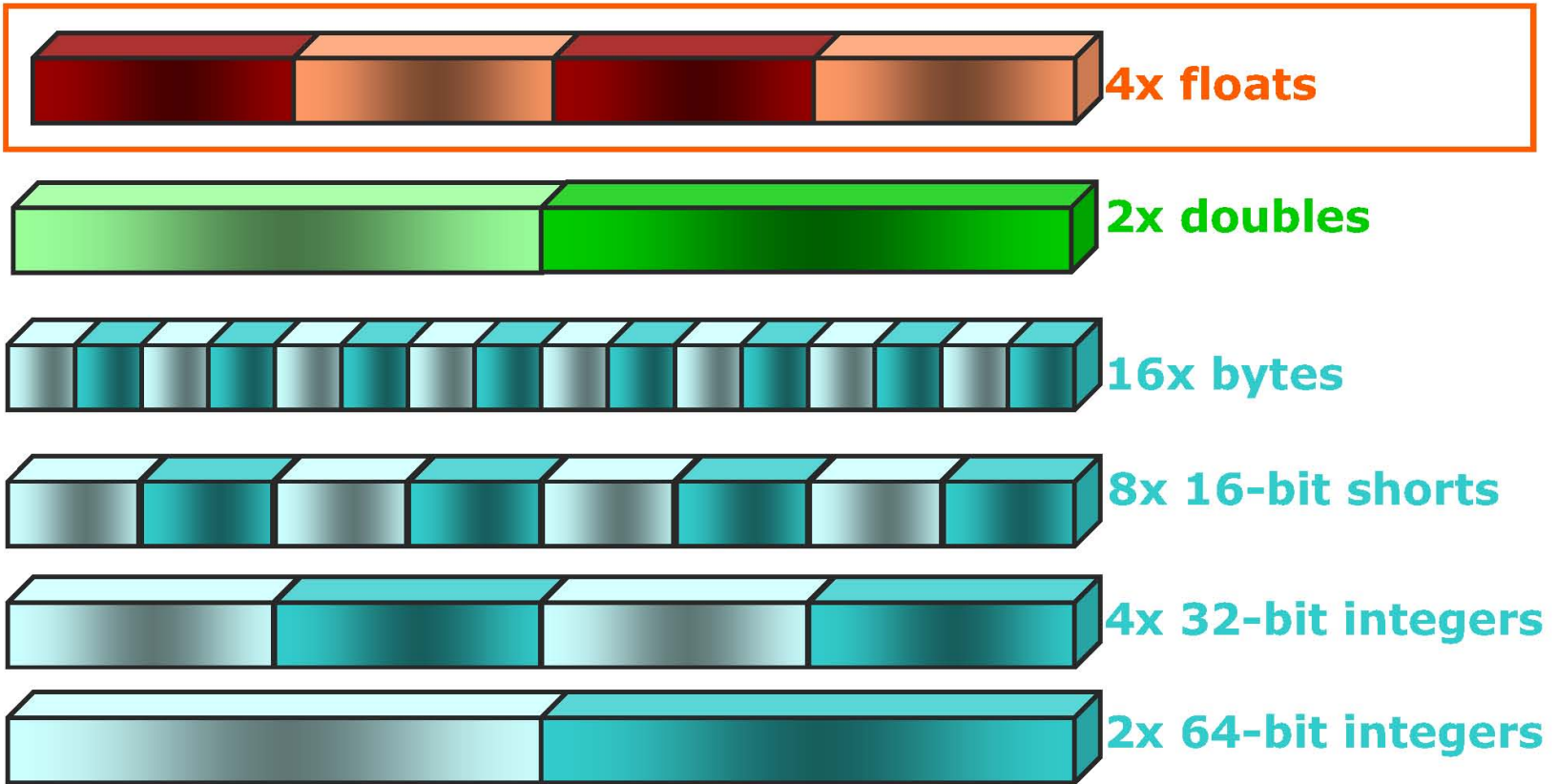


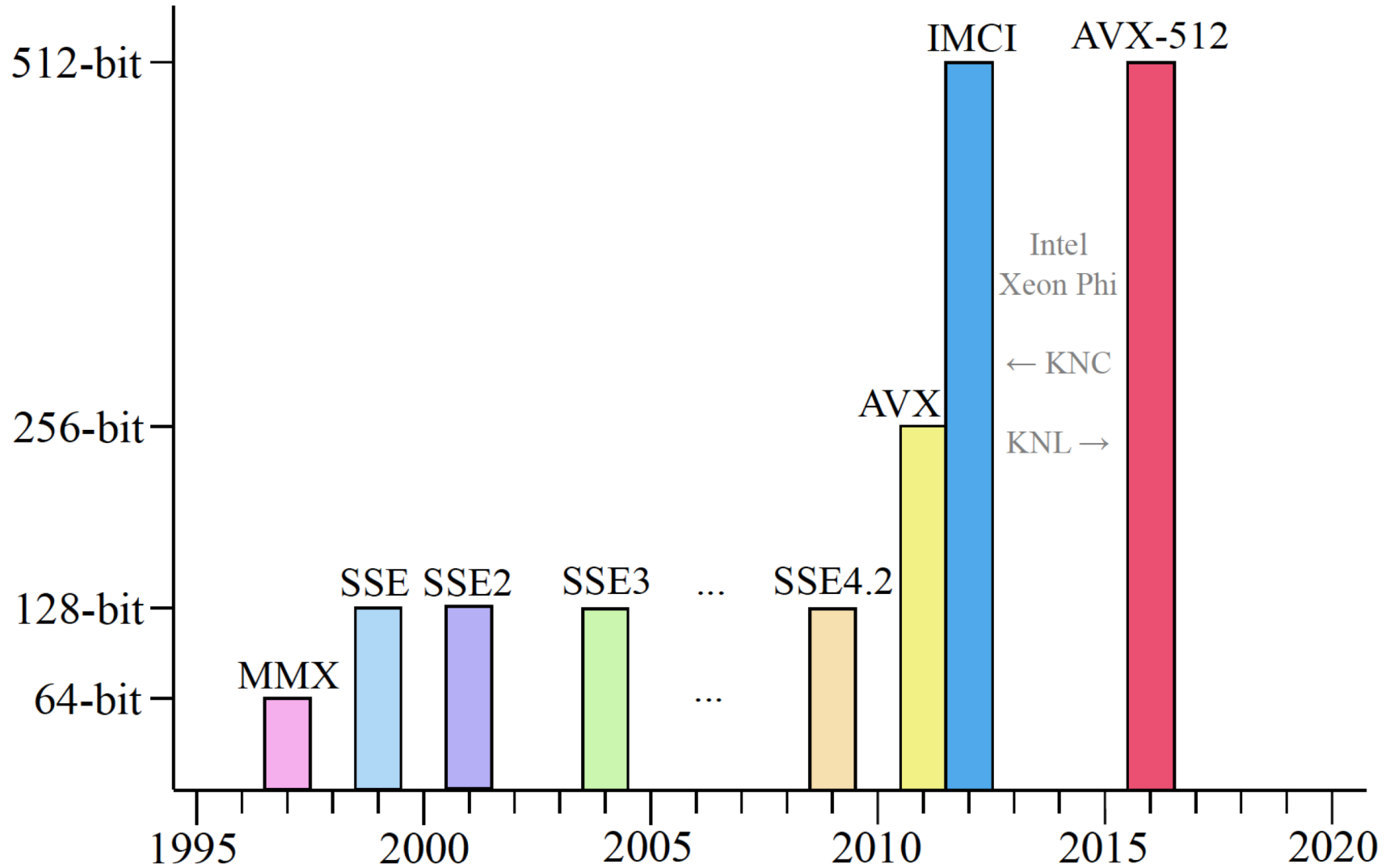
Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation



# SSE Data types

128 bit word (XMM)





# 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;
```

```
typedef float __attribute__((vector_size(16))) Vec4F;  
Vec4F a{0,1f,-2f,3f}, b{-1f,-2f,3f,0}, c{0,2.f,4.f,5.f}, zero{0};
```

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


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

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

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

```
    □ scalar code: if(a>pi/8.f) t= (t-1.0f)/(t+1.0f) ;
```

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



**always computed**



---

# “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 pragma
- OpenMP4
  - New: implementations still experimental
  - icc: “fully supported?”
  - gcc: syntax and code generation ok, vectorization relies on the corresponding compiler pass
- OpenCL:
  - Uses compiler backend

# Loop 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/4; ; i+=4){  
    V4& va = *((V4*)a[i]);  
    va += *((V4 const *)b[i]);  
} vectorized loop (4 times less iterations)
```

◇ loop in vector notation:

```
◇ for (i=0; i<(N-N%4); i+=4){  
    V4& va = *((V4*)a[i]);  
    va += *((V4 const *)b[i]);  
} vectorized loop
```

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

◇ Loop based vectorization

◇ No dependences between iterations

```

1 #include<array>
2 float add(float x,float y) { return x+y;}
3
4 typedef float __attribute__( ( vector_size( 16 ) ) ) Vec4F;
5 Vec4F add(Vec4F x, Vec4F y){ return x+y;}
6
7 typedef float __attribute__( ( vector_size( 32 ) ) ) Vec8F;
8 Vec8F add(Vec8F x, Vec8F y){ return x+y;}
9
10 typedef double __attribute__( ( vector_size( 32 ) ) ) Vec4D;
11 Vec4D add(Vec4D x, Vec4D y){ return x+y;}
12
13 using V4 = std::array<float,4>;
14 void add(V4 const & x, V4 const & y, V4 & z){
15     #pragma GCC ivdep
16     for (int i=0; i<4;++i) z[i]=x[i]+y[i];
17 }
18
19 void add(float const * __restrict__ x, float const * __restrict__ y, float * __restrict__ z, int N)
20     for (int i=0; i<N;++i) z[i]=x[i]+y[i];
21 }
  
```

```

x86-64 gcc 9.2 (Editor #1, Compiler #1) C++
x86-64 gcc 9.2 -Wall -O3 -std=c++17
A [ ] LTO: [ ] libf: [x] .text [x] // [x] \s+ [ ] Intel [x] Demangle
Libraries + Add new... Add tool...
1 add(float, float):
2     addss %xmm1, %xmm0
3     ret
4 add(float __vector(4), float __vector(4)):
5     addps %xmm1, %xmm0
6     ret
7 add(float __vector(8), float __vector(8)):
8     movaps 8(%rsp), %xmm1
9     movaps 24(%rsp), %xmm2
10    movq %rdi, %rax
11    addps 40(%rsp), %xmm1
12    addps 56(%rsp), %xmm2
13    movaps %xmm1, (%rdi)
14    movaps %xmm2, 16(%rdi)
15    ret
16 add(double __vector(4), double __vector(4)):
17    movapd 8(%rsp), %xmm1
18    movapd 24(%rsp), %xmm2
19    movq %rdi, %rax
20    addpd 40(%rsp), %xmm1
21    addpd 56(%rsp), %xmm2
22    movaps %xmm1, (%rdi)
23    movaps %xmm2, 16(%rdi)
24    ret
25 add(std::array<float, 4ul> const&, std::array<float, 4ul> const&,
26     movups (%rdi), %xmm0
27     movups (%rsi), %xmm1
28     addps %xmm1, %xmm0
29     movups %xmm0, (%rdx)
30     ret
31 add(float const*, float const*, float*, int):
32     testl %ecx, %ecx
33     jle .L7
34     leal -1(%rcx), %eax
35     cmpl $2, %eax
36     jbe .L12
37     movl %ecx, %r8d
38     xorl %eax, %eax
39     shrl $2, %r8d
40     salq $4, %r8
41 .L10:
42     movups (%rdi,%rax), %xmm0
43     movups (%rsi,%rax), %xmm1
44     addps %xmm1, %xmm0
45     movups %xmm0, (%rdx,%rax)
46     addq $16, %rax
47     cmpq %r8, %rax
48     jne .L10
49     movl %ecx, %eax
50     andl $-4, %eax
51     testb $3, %cl
52     je .L14
  
```



```

1 #include<array>
2 float add(float x,float y) { return x+y;}
3
4 typedef float __attribute__( ( vector_size( 16 ) ) ) Vec4F;
5 Vec4F add(Vec4F x, Vec4F y){ return x+y;}
6
7 typedef float __attribute__( ( vector_size( 32 ) ) ) Vec8F;
8 Vec8F add(Vec8F x, Vec8F y){ return x+y;}
9
10 typedef double __attribute__( ( vector_size( 32 ) ) ) Vec4D;
11 Vec4D add(Vec4D x, Vec4D y){ return x+y;}
12
13 using V4 = std::array<float,4>;
14 void add(V4 const & x, V4 const & y, V4 & z){
15     #pragma GCC ivdep
16     for (int i=0; i<4;++i) z[i]=x[i]+y[i];
17 }
18
19 void add(float const * __restrict__ x, float const * __restrict__ y, float * __restrict__ z, int N)
20     for (int i=0; i<N;++i) z[i]=x[i]+y[i];
21 }

```

```

1 add(float, float):
2     vaddss %xmm1, %xmm0, %xmm0
3     ret
4 add(float __vector(4), float __vector(4)):
5     vaddps %xmm1, %xmm0, %xmm0
6     ret
7 add(float __vector(8), float __vector(8)):
8     vaddps %ymm1, %ymm0, %ymm0
9     ret
10 add(double __vector(4), double __vector(4)):
11     vaddpd %ymm1, %ymm0, %ymm0
12     ret
13 add(std::array<float, 4ul> const&, std::array<float, 4ul> const&,
14     vmovups (%rsi), %xmm0
15     vaddps (%rdi), %xmm0, %xmm0
16     vmovups %xmm0, (%rdx)
17     ret
18 add(float const*, float const*, float*, int):
19     testl %ecx, %ecx
20     jle .L15
21     leal -1(%rcx), %eax
22     cmpl $6, %eax
23     jbe .L12
24     movl %ecx, %r8d
25     xorl %eax, %eax
26     shrl $3, %r8d
27     salq $5, %r8
28 .L10:
29     vmovups (%rdi,%rax), %ymm1
30     vaddps (%rsi,%rax), %ymm1, %ymm0
31     vmovups %ymm0, (%rdx,%rax)
32     addq $32, %rax
33     cmpq %r8, %rax
34     jne .L10
35     movl %ecx, %eax
36     andl $-8, %eax
37     testb %7, %cl
38     je .L16
39     vzeroupper
40 .L9:
41     movslq %eax, %r8
42     vmovss (%rdi,%r8,4), %xmm0
43     vaddss (%rsi,%r8,4), %xmm0, %xmm0
44     vmovss %xmm0, (%rdx,%r8,4)
45     leal 1(%rax), %r8d
46     cmpl %r8d, %ecx
47     jle .L15
48     movslq %r8d, %r8
49     vmovss (%rdi,%r8,4), %xmm0
50     vaddss (%rsi,%r8,4), %xmm0, %xmm0
51     vmovss %xmm0, (%rdx,%r8,4)
52     leal 2(%rax), %r8d
53     cmpl %r8d, %ecx

```

# Loop Dependences

```
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
```

```
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;  
}
```

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**

# Reduction

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

```
// pseudo code  
float innerProduct() {  
    V4 s ={0};  
    for (int i=0; i<N/4; ++i)  
        s += *((V4 const *)a[i]) * *((V4 const *)b[i]);  
  
    // horizontal sum;  
    float sum=s[0];  
    for (int j=1;j!=4;++j)sum+=s[j];  
    return sum;  
}
```

Requires *relaxed* float-math rules  
(-Ofast)

# Reduction

$$S = X_0 + X_1 + X_2 + X_3 + X_4$$

$$X_0 + X_1 + X_2 + X_3$$

+

$$X_4 + X_5 + X_6 + X_7$$

+

$$X_8 + X_9 + X_{10} + X_{11}$$

=

$$S_0 + S_1 + S_2 + S_3$$

=

$$S$$

# Real code <http://goo.gl/3ku9EJ>

```
float a[1024],b[1024]; int N=1024;
float innerProduct() {
    float s=0;
    for (int i=0; i!=N; i++)
        s+= a[i]*b[i];
    return s;
}
```

.L3:

```
movss    a(%rax), %xmm1
addq     $4, %rax
mulss   b-4(%rax), %xmm1
addss   %xmm1, %xmm0
cmpq    %rdx, %rax
jne     .L3
```

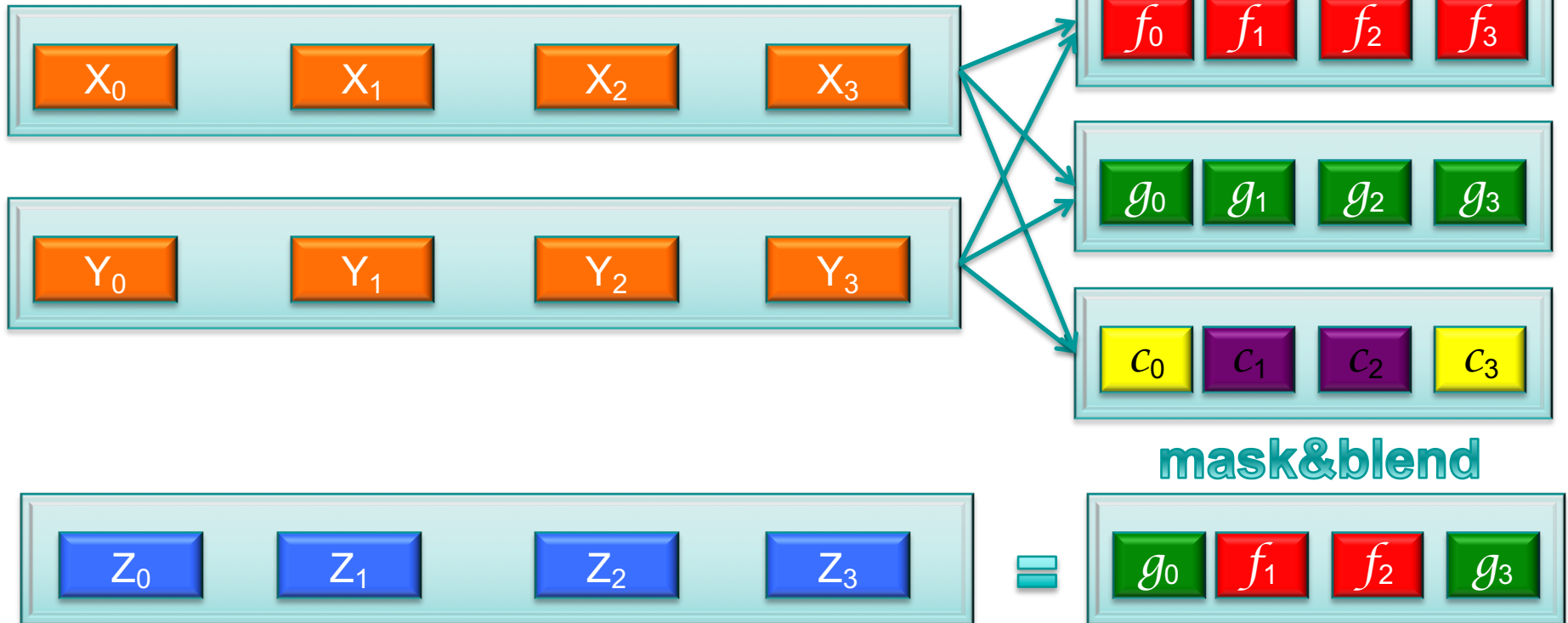
.L4:

```
movaps  b(%rsi), %xmm1
addl    $1, %edi
addq    $16, %rsi
mulps   a-16(%rsi), %xmm1
addps   %xmm1, %xmm0
cmpl    %edi, %edx
ja      .L4
haddps  %xmm0, %xmm0
haddps  %xmm0, %xmm0
cmpl    %ecx, %eax
je      .L15
```

# Conditional Code

$$Z_i = c(X_i, Y_i) ? f(X_i, Y_i) : g(X_i, Y_i)$$

Example:  $z = x > y ? x/y : y/x;$



# Conditional code : <https://godbolt.org/g/shCUuE>

```
double x[1024],y[1024],z[1024]; int N=1024;
void foo() {
    for (int i=0; i!=N; i++) {
        if (x[i]>y[i]) z[i]=x[i]/y[i];
        else z[i]=y[i]/x[i];
    } // exercise: write it with only one division
}
```

loop in vector notation:

```
for (i=0; i<N/4; ++i){
    V4 vx = &((V4 const *)x[i]);
    V4 vy = &((V4 const *)y[i]);
    V4 & vz = *((V4 *)z[i]);
    I4 t = vx > vy;
    // compute both branches
    auto a = vx / vy; auto b = vy / vx;
    // mask and “blend”
    vz = t ? a : b;
}
```

```
foo():
    xorl    %eax, %eax
.L2:
    movapd  x(%rax), %xmm2
    addq    $16, %rax
    movapd  y-16(%rax),
    %xmm0
    movapd  %xmm2, %xmm3
    divpd   %xmm0, %xmm3
    movapd  %xmm0, %xmm1
    cmpltpd %xmm2, %xmm0
    divpd   %xmm2, %xmm1
    blendvpd %xmm0, %xmm3,
    %xmm1
    movaps  %xmm1, z-
    16(%rax)
    cmpq    $8192, %rax
    jne    .L2
    ret
```

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

# Vectorization of “math function”

- Currently linux libm does not support vectorization
  - NEW: limited set of vector functions in libc >2.21
- Exploit compiler + vendor libraries
  - Requires licensed libs by intel and/or amd or libc > 2.21
  - No change in user code: just recompile
    - `-mveclibabi=svml -L/.../lib/intel64 -lsvml -lirc`
    - `-mveclibabi=acml -L/.../lib/amd64 -lacml -lamdlibm`
    - `-fopenmp` (if libc > 2.21)
- Exploit auto-vectorization
  - Modified cephess library (or other open source)
    - Requires header-file + different function names
  - Look into vdt/ (<https://svnweb.cern.ch/trac/vdt>)
    - Project for a student: port vdt to gcc simd-vectors
  - For your exercise use SimpleSinCos.h
    - `simpleSin(x)`, `simpleCos(x)`, `simpleSinCos(x,s,c)`



---

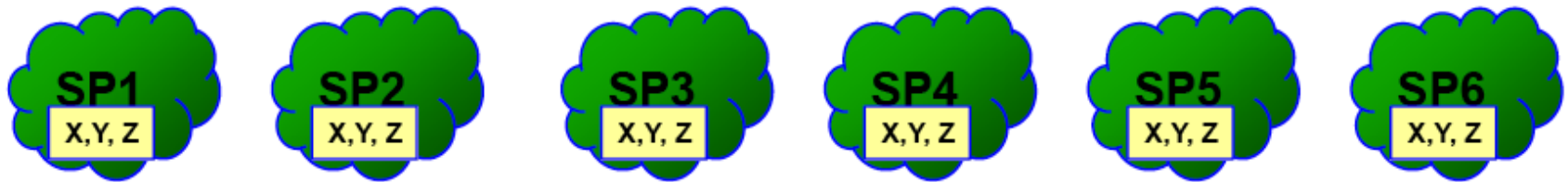
<https://github.com/CppCon/CppCon2014/tree/master/Presentations/Data-Oriented%20Design%20and%20C%2B%2B>

<http://aras-p.info/texts/files/2018Academy%20-%20ECS-DoD.pdf>

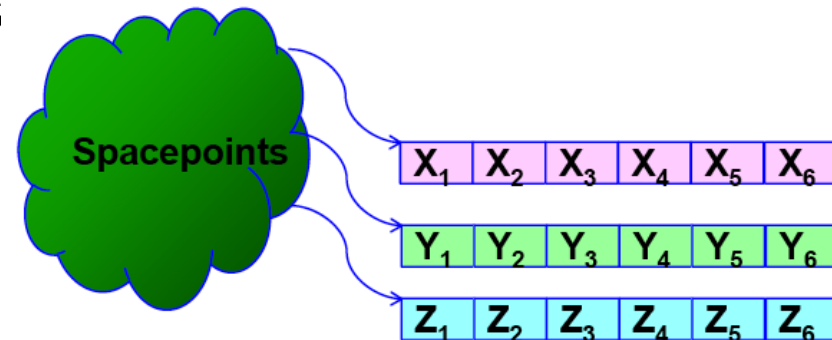
# DATA ORGANIZATION

# Data Organization: AoS vs SoA

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



- Difficult to fit stream computing
- Better to use a Structure of Arrays
  - (column-wise storage)
- 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**)



# AoS

vs

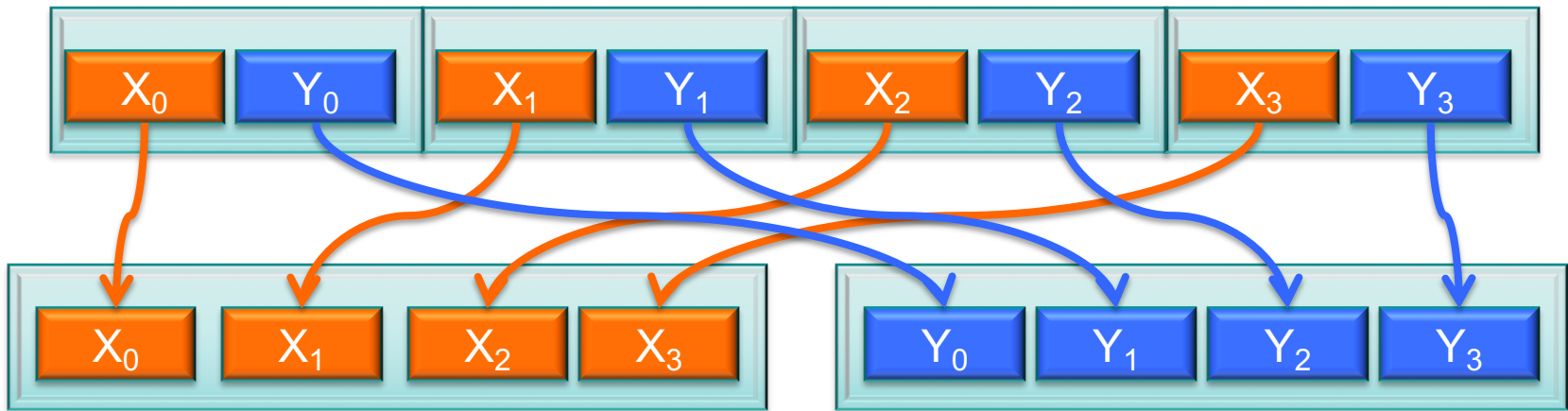
# SoA

```
struct Point {  
    float x,y,z;  
};  
struct Points {  
    std::vector<Point> p;  
};
```



```
struct Points {  
    std::vector<float> x,y,z;  
};
```

The choice of AoS vs SoA depends on the access patterns in the application  
Operations on AoS can be vectorized at the cost of *shuffles* or *permutations*  
For large structures this becomes not profitable pretty fast.  
In general SoA shall be preferred to fully exploit vectorization

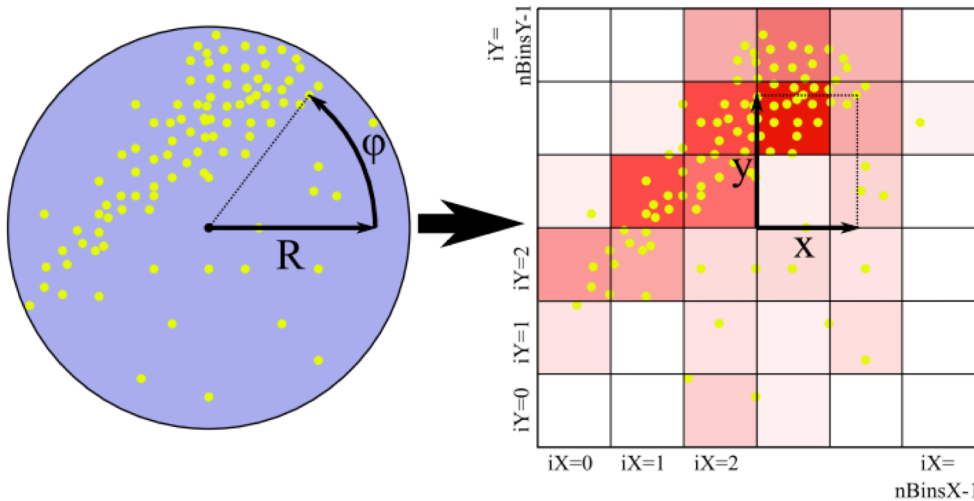


# Don't stop the stream!

- 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

# Excercise

<https://infn-esc.github.io/esc16/vectorization/binning.html>



Take points in polar coordinates  
Fill a two dimensional  
cartesian Histogram

```
struct Point { float phi,r; };
struct Points {
    std::array<Point,N> p;
};
Points points;
constexpr int NBin = 100;
struct Hist {
    int bin[NBin+1][NBin+1] = {0};
};
float binwidth = 2./NBin;
Hist h;
```

```
For(auto const & p : points.p) {
    auto x = p.r*std::cos(p.phi);
    auto y = p.r*std::sin(p.phi);
    int xbin = (x+1.f)/binwidth;
    int ybin = (y+1.f)/binwidth;
    assert(xbin>=0 && ybin>=0);
    assert(xbin<101 && ybin<101);
    ++h.bin[xbin][ybin];
}
```

# Exercise 1

- Open SimpleVectorization.cc  
<http://goo.gl/SAiici>
- 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

<https://infn-esc.github.io/esc18/vectorization/pi.html>

- Open pi.cpp <http://goo.gl/zl3WqS>
  - Compile it with -O2, -Ofast, more fancy options
  - Change double in float
  - Try to vectorize using native vectors...

# Pi Program: Vectorization with intrinsics (SSE)



See <https://software.intel.com/sites/landingpage/IntrinsicsGuide/>

```
float pi_sse(int num_steps)
{ float scalar_one =1.0, scalar_zero = 0.0, ival, scalar_four =4.0, step, pi, vsum[4];
  step = 1.0/(float) num_steps;

  __m128 ramp = _mm_setr_ps(0.5, 1.5, 2.5, 3.5);
  __m128 one = _mm_load1_ps(&scalar_one);
  __m128 four = _mm_load1_ps(&scalar_four);
  __m128 vstep = _mm_load1_ps(&step);
  __m128 sum = _mm_load1_ps(&scalar_zero);
  __m128 xvec; __m128 denom; __m128 eye;

  for (int i=0;i< num_steps; i=i+4){ // unroll loop 4 times
    ival = (float)i; // and assume num_steps%4 = 0
    eye = _mm_load1_ps(&ival);
    xvec = _mm_mul_ps(_mm_add_ps(eye,ramp),vstep);
    denom = _mm_add_ps(_mm_mul_ps(xvec,xvec),one);
    sum = _mm_add_ps(_mm_div_ps(four,denom),sum);
  }
  _mm_store_ps(&vsum[0],sum);
  pi = step * (vsum[0]+vsum[1]+vsum[2]+vsum[3]);
  return pi;
} // credit T.M.
```