Vincenzo Innocente:

"Optimal floating point computation"

- IEEE 754 standard
- Expression optimization
- Approximate Math
- Vector Algebra using SIMD instructions
- Loop vectorization using SIMD instructions

Prepare for the excercises

```
tar -zxf /storage/software/innocente/Bertinoro2010.tgz
cd Bertinoro2010/
source install.sh
echo $CXX
echo $CXXFLAGS
echo $OPTFLAGS
which c++
$CXX $CXXFLAGS examples/paranoia.c; ./a.out
 icpc examples/paranoia.c ; ./a.out
```

Floating behaviour…

```
void f() {
  int n=0; float w=1; float y=0;
  do {
    y = w++;
    ++n;
  } while (w>y);
  std::cout << n << " " << w << std::endl;
}
void i() {
  int w=0; int y=0;
  do { y = w++; } while (w>y);
  std::cout << w << std::endl;
}
void f2() {
  float x = 1.f/3.f;
  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==x) << " " << (z<x) << " " << (z>x) << std::endl;
}
```

Disclaimer, caveats, references

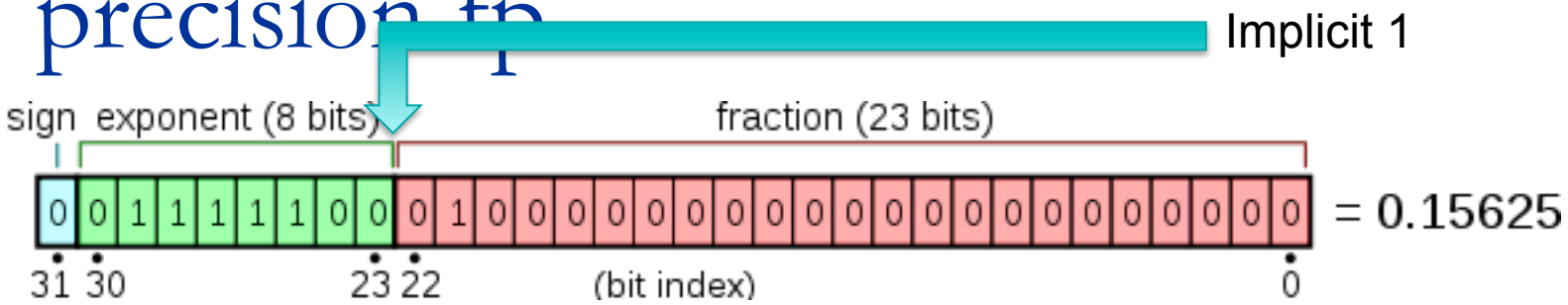
- This is NOT a full overview of IEEE 754
- It applies to x86_64 systems and gcc > 4.5.0
  - Other compilers have similar behavior, details differ
- General References
  - Wikipedia has excellent documentation on the IEEE 754, math algorithms and their computational implementations
  - Handbook of Floating-Point Arithmetic (as google book)
  - Kahan home page
  - Jeff Arnolds seminar and course at CERN
  - INTEL doc and white papers
  - Ulrich Drepper recent talks

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}

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)	

Examples of single precision fp

3f80 0000 = 1

c000 0000 = −2

7f7f ffff ≈ 3.4028234×10^{38} (max single precision)

0000 0000 = 0

8000 0000 = −0

7f80 0000 = infinity

ff80 0000 = -infinity

7f80 c000 = NaN

3eaa aaab ≈ 1/3

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

Look into a float

```
void look(float 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;
 }
```

Floating Point Math

- Floating point numbers are NOT real numbers
  - They exist in a finite number (~2^{32})
  - Exist a “next” and a “previous”
    - 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
    - $x + \varepsilon - x \neq \varepsilon$ (can be easily 0 or ∞)
  - Their algebra in not associative
    - $(a+b) + c \neq a+(b+c)$
    - $a/b \neq a*(1/b)$
    - $(a+b)*(a-b) \neq a^2 - b^2$

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

Improve accuracy

- Operations among fp can suffer of loss of accuracy and *catastrophic* cancellations
  - $x + \varepsilon - x \neq \varepsilon$
- Accuracy can be improved
  - Refactoring: x^2-y^2 ➔ $(x+y)*(x-y)$
  - Summing numbers in order (smaller first)
  - Summing negative and positive separately
  - Compensating for the numerical error (see next slides)

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;
```

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);
}
```

COMPILER OPTIMIZATION OPTIONS

MOTIVATIONS

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 = localP.mag();
double normalisedPath = fabs(p/localP.z())*materialConstants.radLen();
double z = exp(-normalisedPath);
double varz = (exp(-normalisedPath*log(3.)/log(2.))- exp(-2*normalisedPath));
```

```
double pt = mom.transverse();
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));
```

Formula Translation: is this still ok?

```
// Energy loss and variance according to Bethe and Heitler, see also
 // Comp. Phys. Comm. 79 (1994) 157.
 //
double p = localP.mag();
double normalisedPath = std::abs(p/localP.z())*materialConstants.radLen();
double z = exp(-normalisedPath);
double varz = exp(-normalisedPath*log3o2)-z*z;
```

```
double pt2i = 1/mom.perp2();
double j = a_i*((d_x * mom.x() + d_y * mom.y())*pt2i);
double r_x = d_x – mom.x()*(2*((d_x*mom.x()+d_y*mom.y())*pt2i));
double r_y = d_y – mom.y()*(2*((d_x*mom.x()+d_y*mom.y())*pt2i));
double s = pt2i/sqrt(1 - j*j);
```

Can we let the compiler do it for us?

Optimization options beyond O2

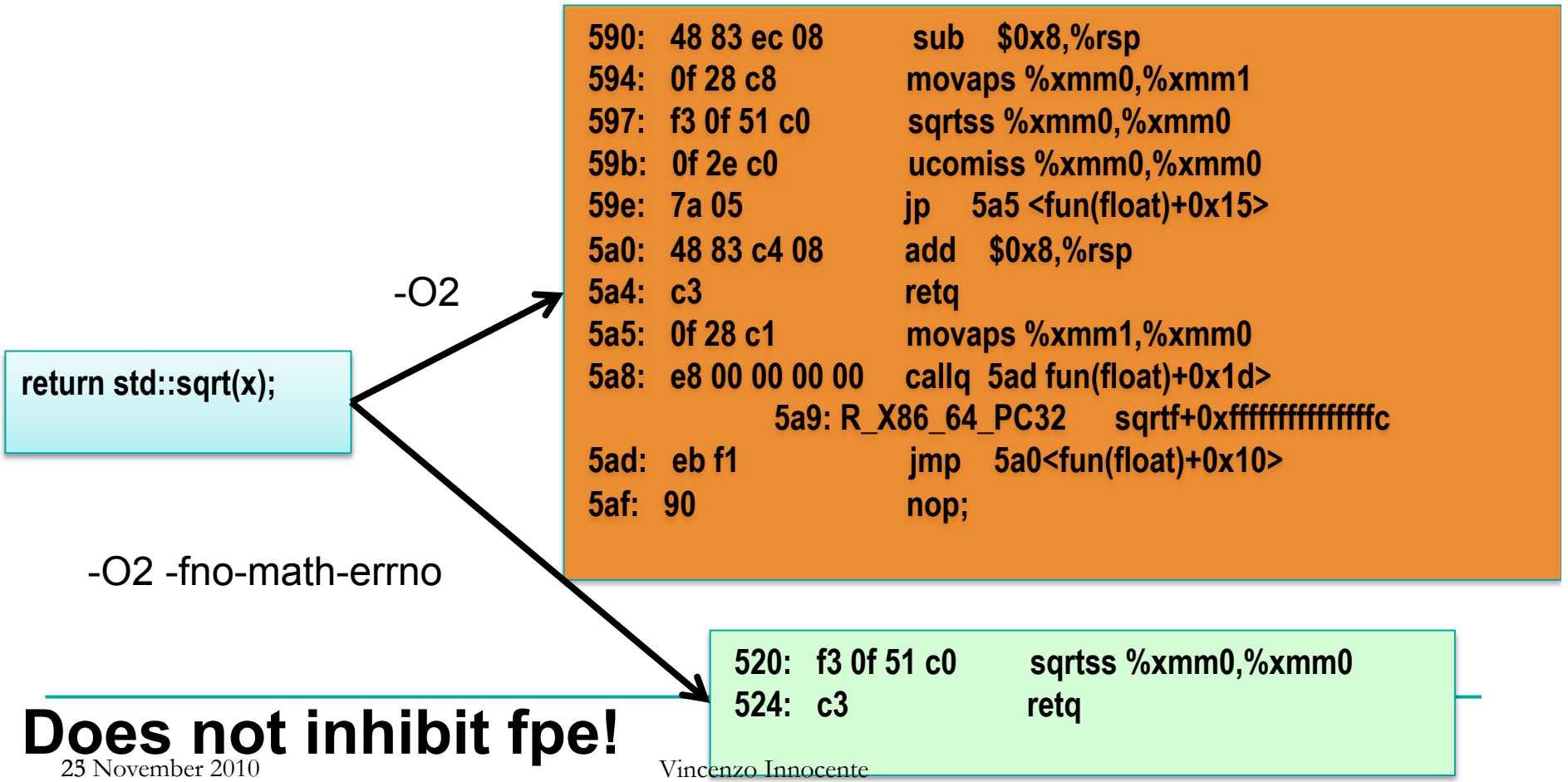
GCC is VERY conservative: standard optimization guarantees full regression and full conformance to standards including

- Error setting as in C stdlib
- IEEE 754

- We do not necessary rely on all the details of these standards
  - Who checks “errno”?
  - In general our math is “finite” and we do not “explicitly” rely on proper treatment of infinities, subnormals, signed zeros
- To allow the compiler to perform more aggressive optimization of math expressions we need to relax part of the conformance to standards
- There are three ways to specify optimization options
  - On the command line
  - As pragma (push and pop)
  - As function attribute
  - **NEW**: *-Ofast* introduced in GCC 4.6 (most probably equivalent to *–O3 –ffast-math*)

-fno-math-errno

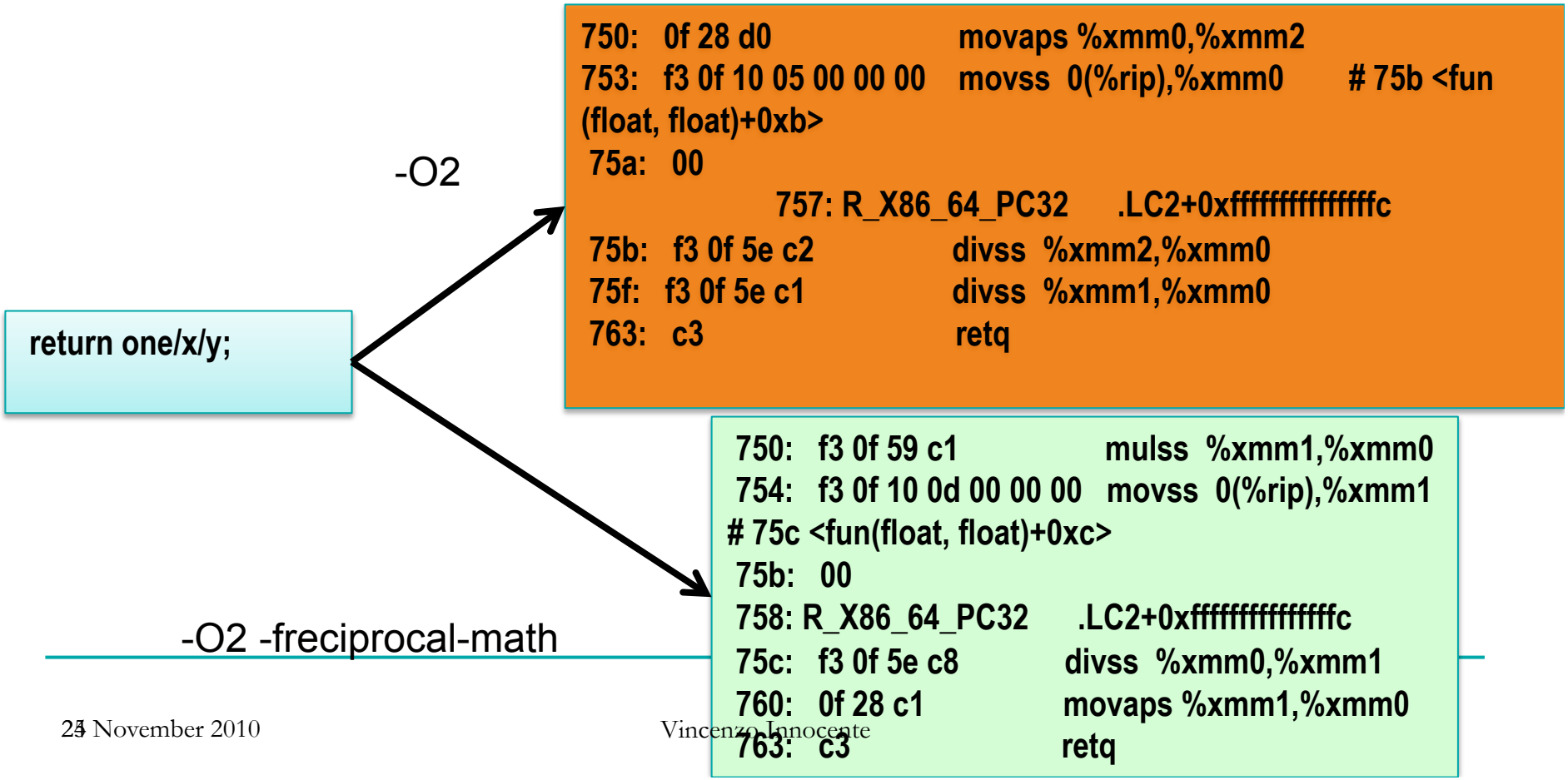
Do not set ERRNO after calling math functions that are executed with a single instruction, e.g., sqrt. A program that relies on IEEE exceptions for math error handling may want to use this flag for speed while maintaining IEEE arithmetic compatibility.



Does not inhibit fpe!

-freciprocal-math

*Allow the reciprocal of a value to be used instead of dividing by the value if this enables optimizations. For example "x /y" can be replaced with "x * (1/y)" which is useful if "(1/y)" is subject to common subexpression elimination. Note that this loses precision and increases the number of flops operating on the value.*

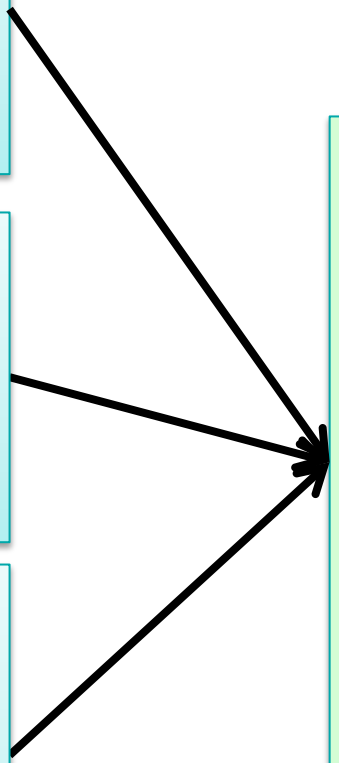


-fassociative-math

```
float multiassoc1(float x,float y) {
 float a = x + one + y;
 float b = x + two + y;
 return a/b;
}
```

```
float multiassoc2(float x,float y) {
 float a = x + (one + y);
 float b = x + (two + y);
 return a/b;
}
```

```
float multiassoc3(float x,float y) {
 float a = one + (x + y);
 float b = two + (x + y);
 return a/b;
}
```



```
 1d0:   f3 0f 58 c1           addss  %xmm1,%xmm0
 1d4:   f3 0f 10 0d 00 00 00    movss  0(%rip),%xmm1
# 1dc <multiassoc1(float, float)+0xc>
 1db:   00
 1d8: R_X86_64_PC32      .LC2+0xfffffffffffffffc
 1dc:   f3 0f 58 c8           addss  %xmm0,%xmm1
 1e0:   f3 0f 58 05 00 00 00    addss  0(%rip),%xmm0
# 1e8 <multiassoc1(float, float)+0x18>
 1e7:   00
1e4: R_X86_64_PC32      .LC5+0xfffffffffffffffc
 1e8:   f3 0f 5e c8           divss  %xmm0,%xmm1
 1ec:   0f 28 c1              movaps %xmm1,%xmm0
 1ef:   c3                    retq
```

-fassociative-math

Allow re-association of operands in series of floating-point operations. This violates the ISO C and C++ language standard by possibly changing computation result.
*NOTE: re-ordering may change the sign of zero as well as ignore NaNs and inhibit or create underflow or overflow (and thus cannot be used on a code which relies on rounding behavior like "(x + 2**52) -2**52)". May also reorder floating-point comparisons and thus may not be used when ordered comparisons are required. This option requires that both -fno-signed-zeros and -fno-trapping-math be in effect. Moreover, it doesn't make much sense with -frounding-math. For Fortran the option is automatically enabled when both -fno-signed-zeros and -fno-trapping-math are in effect.*

-fassociative-math (side effects)

```
float kloop() {
 float y=0;
 float w = one;
 do  {
   w = w + w;
   y = w + one;
   float z = y - w;
   y = z - one;
 } while (minusOne + std::abs(y) < zero);
 /*.. now W is just big enough that
    |((W+1)-W)-1| >= 1 ...*/
 return w;
}
```

```
0000000000000150 <kloop()>:
150:   eb fe                jmp    150 <kloop()>
152:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
159:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

```
movss  0(%rip),%xmm0        # 1c8 <f(float)+0x8>
R_X86_64_PC32      .LC4+0xfffffffffffffffc
retq
```

```
return x + eps –x;
```

-fno-signed-zeros & -ffinite-math-only

```
float pluszero(float x) {
  return x + zero;
}
float prodzero(float x) {
  return x*zero;
}
float divzero(float x) {
  return zero/x;
}
```

```
0000000000000220 <pluszero(float)>:
 220:        f3 c3                   repz retq
0000000000000230 <prodzero(float)>:
 230:        f3 0f 59 05 00 00 00    mulss  0(%rip),%xmm0
# 238 <prodzero(float)+0x8>
 237:        00
             234: R_X86_64_PC32   .LC4+0xfffffffffffffffc
 238:        c3                      retq
0000000000000240 <divzero(float)>:
 240:        0f 28 c8                movaps %xmm0,%xmm1
 243:        0f 57 c0                xorps  %xmm0,%xmm0
 246:        f3 0f 5e c1             divss  %xmm1,%xmm0
 24a:        c3                      retq
```

-ffinite-math-only

```
00000000000001d0 <pluszero(float)>:
 1d0:   f3 c3                repz retq
00000000000001e0 <prodzero(float)>:
 1e0:   0f 57 c0             xorps  %xmm0,%xmm0
 1e3:   c3                   retq
00000000000001f0 <divzero(float)>:
 1f0:   0f 57 c0             xorps  %xmm0,%xmm0
 1f3:   c3                   retq
```

-fno-trapping-math

Compile code assuming that floating-point operations cannot generate user-visible traps. These traps include division by zero, overflow, underflow, inexact result and invalid operation. This option requires that -fno-signaling-nans be in effect. Setting this option may allow faster code if one relies on "non-stop" IEEE arithmetic, for example.

It does not inhibits fpe in itself:

It allows optimization that may not produce fpe anymore such as *for (x + eps –x)* when *eps* is close to *inf*

What the compiler does not do (yet)

```
float sqrt5(float x, float y) {
  float p = std::sqrt(x*x+y*y);
  float a = x/p/p;
  float b = y/p/p;
  return a+b;
}
```

```
0000000000000540 <sqrt5(float, float)>:
540:   0f 28 d1          movaps %xmm1,%xmm2
543:   f3 0f 59 c9       mulss  %xmm1,%xmm1
547:   f3 0f 58 d0       addss  %xmm0,%xmm2
54b:   f3 0f 59 c0       mulss  %xmm0,%xmm0
54f:   f3 0f 58 c8       addss  %xmm0,%xmm1
553:   f3 0f 51 c9       sqrtss %xmm1,%xmm1
557:   f3 0f 59 c9       mulss  %xmm1,%xmm1
55b:   f3 0f 5e d1       divss  %xmm1,%xmm2
55f:   0f 28 c2          movaps %xmm2,%xmm0
562:   c3                retq
```

```
float sqrt4(float x) {
  float a = one*std::sqrt(x);
  float b = two/std::sqrt(x);
  return a*b;
}
```

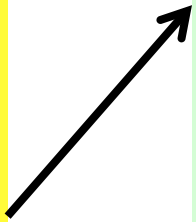
```
000000000000540 <sqrt4(float)>:
540:   f3 0f 51 c8       sqrtss %xmm0,%xmm1
544:   0f 28 c1          movaps %xmm1,%xmm0
547:   f3 0f 58 c1       addss  %xmm1,%xmm0
54b:   f3 0f 5e c1       divss  %xmm1,%xmm0
54f:   c3
```

Pitfalls: undesired conversion

By default in C all floating points are double

```
#include<cmath>
float conv(float x) {
  x+=3.14;
  return pow(x,2);
}

float noconv(float x) {
  x+=3.14f;
  return std::pow(x,2.f);
}
```



```
conv(float):
        unpcklps        %xmm0,%xmm0
        pushq   %rbp
        movq    %rsp,%rbp
        cvtps2pd        %xmm0,%xmm0
        addsd   0x00000036(%rip),%xmm0
        leave
        movddup %xmm0,%xmm0
        cvtpd2ps        %xmm0,%xmm0
        unpcklps        %xmm0,%xmm0
        cvtps2pd        %xmm0,%xmm0
        mulsd   %xmm0,%xmm0
        movddup %xmm0,%xmm0
        cvtpd2ps        %xmm0,%xmm0
        ret
        nop
noconv(float):
        addss   0x00000018(%rip),%xmm0
        pushq   %rbp
        movq    %rsp,%rbp
        leave
        mulss   %xmm0,%xmm0
        ret
```

Pitfalls: inlining, redundant calls

- The compiler inlines a function based on a multitude of rules and heuristics
  - (the inline keyword is not enough)
  - To force inline add the function attribute __attribute__ ((always_inline))
- For non inline functions the compiler cannot know that a function does not have side effects and does not use external information
  - To inform the compiler that a function has no side effects and the result depends only on its input add the function attribute __attribute__ ((pure))

Optimization strategy

- Automatic optimization of fp expression is powerful
  - can clobber careful crafted code
  - handles only “relatively easy” cases
- Best conservative practice is to optimize by hand
  - Try aggressive compiler optimization
    - Check speed-up
    - Inspect assembly code
    - Validate results including “edge” cases
  - Modify code to obtain same or better optimization!

APPROXIMATE (FAST) MATH

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
```

Approximate trigonometry

- truncated polynomial expansion
- Look-up table (w/o interpolation!)
- Both are usually valid only for a limited range of the input values ($-\pi < x < \pi$)
- Care should be taken in benchmarking as their real speed depends on the “context”
  - Cpu pipeline
  - Caches and memory latency

Approximate log

- For Log, the polynomial expansion works in a very limited range (and adding more terms does not help)
- Remember: fp representation is already logarithmic!
  - Icsi algorithm: build lookup table of logs of truncated significands
  - This technique can be used to build and interpolate any type of logarithmic tables of functions

```
template<int N> inline float icsi_log(float val,LookupTable<N> const & lookup_table){
  const int n = N; // precision (in bits)
  union { float val;  int x;} tmp;
  tmp.val=val;
  const int log_2 = ((tmp.x >> 23) & 255) - 127; // exponent
  tmp.x &= 0x7FFFFF;                              //  mantissa
  tmp.x = tmp.x >> (23-n);                        //  quantize mantissa
  val = lookup_table[tmp.x];                      //   lookup precomputed value
  return (val + log_2)* 0.69314718f;              //   natural logarithm
}
```

What to retain

- Floating points are not real number
  - Rounding always occur
- Optimization is balancing accuracy vs speed
  - One size does NOT fit all: it is application dependent
- Developer skill is required to find the optimum

What to bring back home

- Reading and reference material!
- Binary coding and encoding of float
- Approximate math code

SIMD: VECTOR ALGEBRA

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

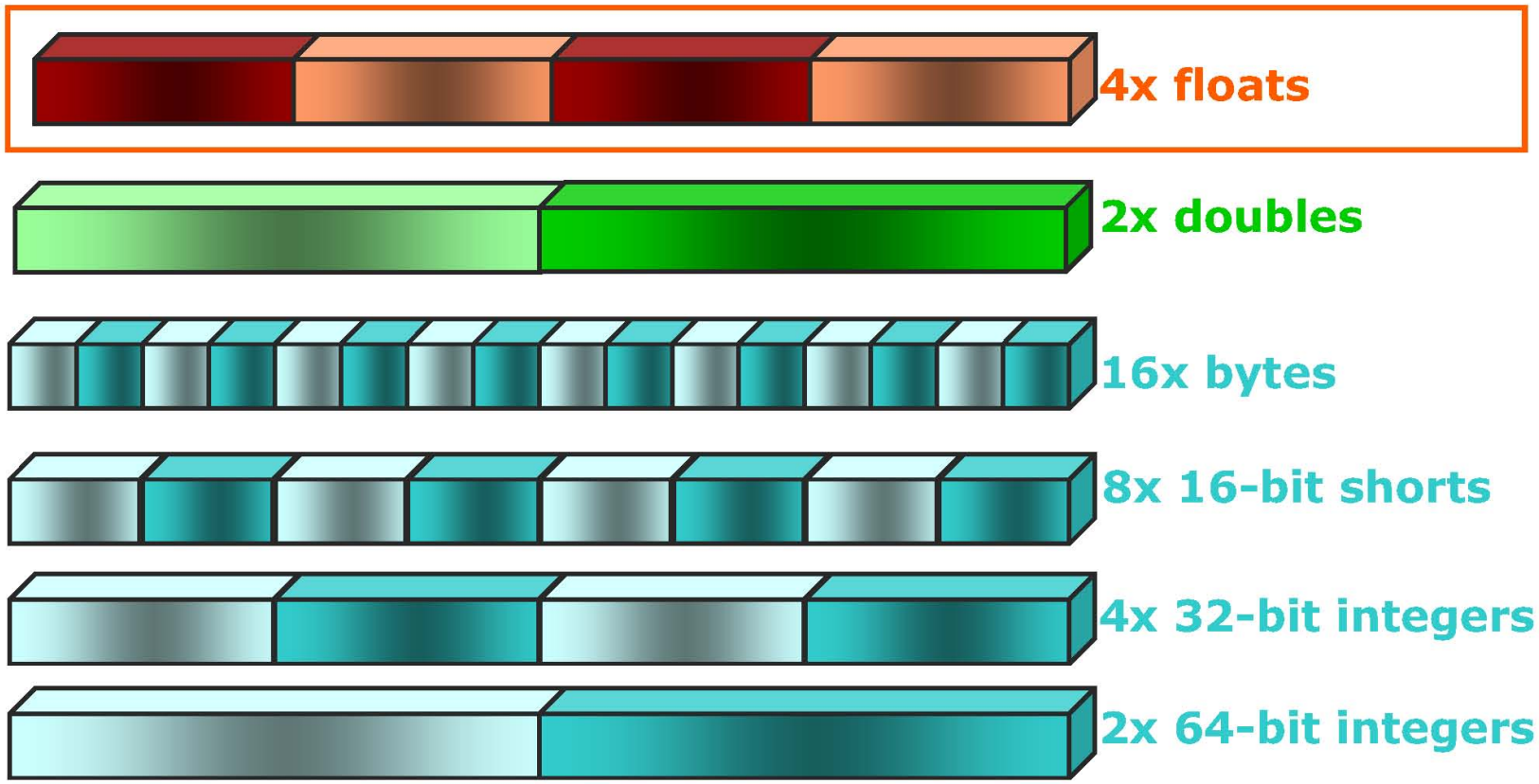
Scalar loop :

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

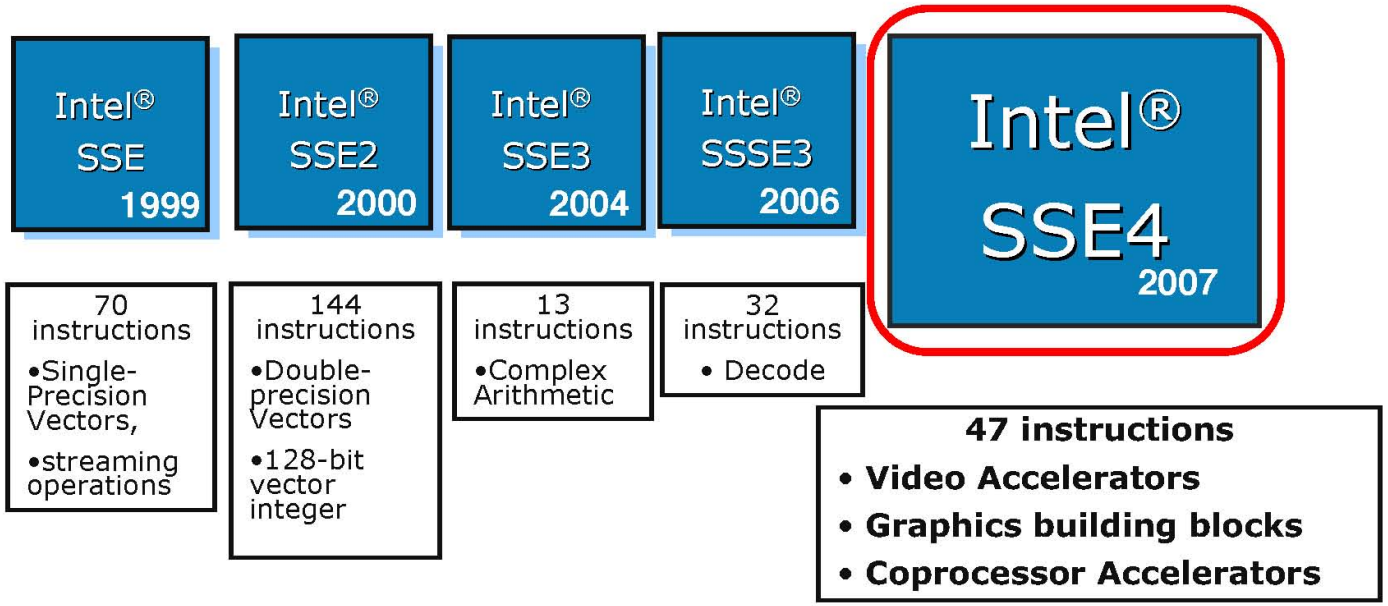
Vector loop :

```
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
```

SSE Data types



Evolution of SSE



SSE instructions overview

- Arithmetic:
  - Multiply, Add, Subtract, Divide, Square root and more.
- Logic:
  - and, and-not, or, xor
- Other:
  - Min/Max , shuffle, packed compares, Blending, type conversion (e.g. int to float and float to double).
- Dedicate functionality:
  - MPSADBW (Fast Block Difference), DPPS (Dot Product).

C Compiler support for SSE – A glance to Intrinsics

- Vector Data Types:
  - __m128 for single precision.
  - __m128i for integers.
  - __m128d for double precision
- Each instructions has its equivalent intrinsic, example intrinsics:
  - extern __m128 _mm_add_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_mul_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_and_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_cmpeq_ps(__m128 _A, __m128 _B);
- More details will follow.

Using compiler intrinsics to generate SSE code (1/3)

- Use Intel® c++ compiler 10.0 for SSE4 support.
- #include <smmintrin.h>
- Data Types:
  - __m128
    Vector type of 4 single precision floating point elements.
  - __m128i
    Vector type of 4 integer elements.
  - __m128d
    Vector type of 2 double precision floating point elements.
- Load/Store:
  - _mm_load_pd/ps/si128 , _mm_loadu_pd/ps/si128
  - _mm_store_pd/ps/si128, _mm_storeu_pd/ps/si128

Using compiler intrinsics to generate SSE code (2/3)

- Casting examples – used for type safety no real instruction is executed.
  - __m128 _mm_castpd_ps(__m128d in);
  - __m128i _mm_castpd_si128(__m128d in);
  - __m128d _mm_castps_pd(__m128 in);
  - __m128i _mm_castps_si128(__m128 in);
  - __m128 _mm_castsi128_ps(__m128i in);
  - __m128d _mm_castsi128_pd(__m128i in);
- Type conversion – Instructions are generated to convert from one representation to the other
  - For example: {1.1, 1.0, 1.0, 1.0} ➔ {1, 1, 1, 1}
  - __m128d _mm_cvtepi32_pd(__m128i a);
  - __m128i _mm_cvtpd_epi32(__m128d a);
  - __m128 _mm_cvtepi32_ps(__m128i a);
  - __m128i _mm_cvtps_epi32(__m128 a);
  - __m128 _mm_cvtpd_ps(__m128d a);
  - __m128d _mm_cvtps_pd(__m128 a);

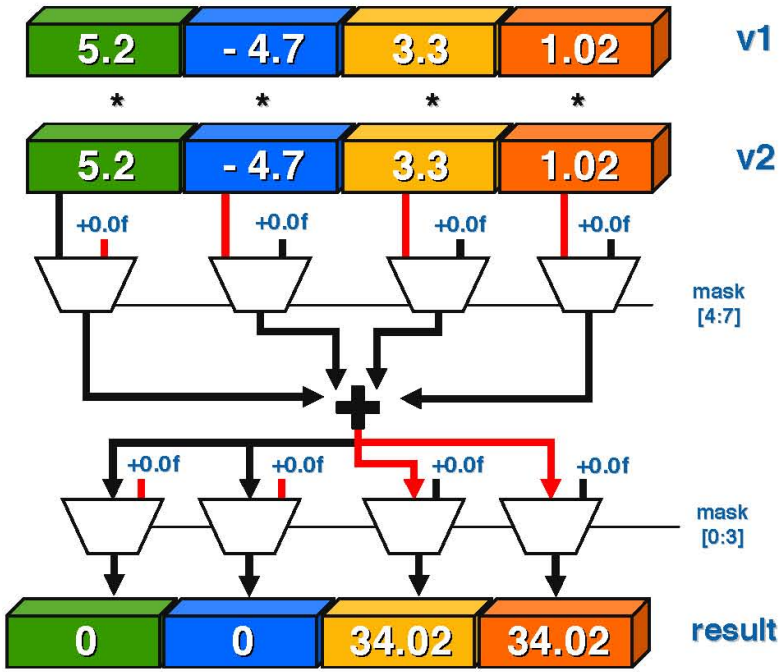
Using compiler intrinsics to generate SSE code (3/3)

- All the instructions have their intrinsic equivalent, example intrinsics:
- Arithmetic:
  - extern __m128 _mm_add_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_sub_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_mul_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_div_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_sqrt_ps(__m128 _A);
  - extern __m128 _mm_min_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_max_ps(__m128 _A, __m128 _B);
- Logical
  - extern __m128 _mm_and_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_andnot_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_or_ps(__m128 _A, __m128 _B);
  - extern __m128 _mm_xor_ps(__m128 _A, __m128 _B);
- Comparison
  - extern __m128 _mm_cmpeq_ps(__m128 _A, __m128 _B);

Dot Product

```
_mm_dp_ps (__m128  val1, __m128  val2, const int mask);
_mm_dp_pd(__m128d val1, __m128d val2, const int mask);
```

result = _mm_dp_ps(v1, v2, 0xF3)



Object Encapsulation of SSE Vectors

```
template<> union Vec4<float> {
    __m128 vec;
    float __attribute__ ((aligned(16))) arr[4];

   Vec4(__m128 ivec) : vec(ivec) {}
   Vec4() { vec = _mm_setzero_ps(); }
   explicit Vec4(float f1) {set1(f1);}
   Vec4(float f1, float f2, float f3, float f4=0) {
     arr[0] = f1; arr[1] = f2; arr[2] = f3; arr[3]=f4;
    }
    void set(float f1, float f2, float f3, float f4=0) {
      vec = _mm_set_ps(f4, f3, f2, f1);
    }
    void set1(float f1) { vec =  _mm_set1_ps(f1); }
    Vec4 get1(unsigned int n) const {
      return _mm_shuffle_ps(vec, vec, _MM_SHUFFLE(n, n, n, n));
    }
};
```

Encapsulating operations

```
inline Vec4F operator+(Vec4F a, Vec4F b) {
  return  _mm_add_ps(a.vec,b.vec);
}
inline Vec4F operator*(float a, Vec4F b) {
  return  _mm_mul_ps(_mm_set1_ps(a),b.vec);
}
inline Vec4F operator-(Vec4F a) {
  const __m128 neg = _mm_set_ps ( -0.0 , -0.0 , -0.0, -0.0);
  return _mm_xor_ps(a.vec,neg);
}
```

```
// at least in gcc this works as well
inline Vec4F operator+(Vec4F a, Vec4F b) {return a.vec+b.vec;}
inline Vec4F operator*(float a, Vec4F b) {
   __m128 va = {a,a,a,a,}; return va*b.vec;
}
inline Vec4F operator-(Vec4F a) {return –a;}
```

Scalar product

```
 //dot   for (i=0; i!=4; ++i) sum+=v1[i]*v2[i];
  inline __m128 _mm_dot_ps(__m128 v1, __m128 v2) {
#ifdef __SSE4_1__
    return _mm_dp_ps(v1, v2, 0xff);
#else
    __m128 mul = _mm_mul_ps(v1, v2);
#ifdef __SSE3__
    mul = _mm_hadd_ps(mul,mul);
    return _mm_hadd_ps(mul,mul);
#else
    __m128 swp = _mm_shuffle_ps(mul, mul, _MM_SHUFFLE(1, 0, 3, 2));
    mul = _mm_add_ps(mul, swp);
    swp = _mm_shuffle_ps(mul, mul, _MM_SHUFFLE(2, 3, 0, 1));
    return _mm_add_ps(mul, swp);
#endif
#endif
  }
```

Vector product

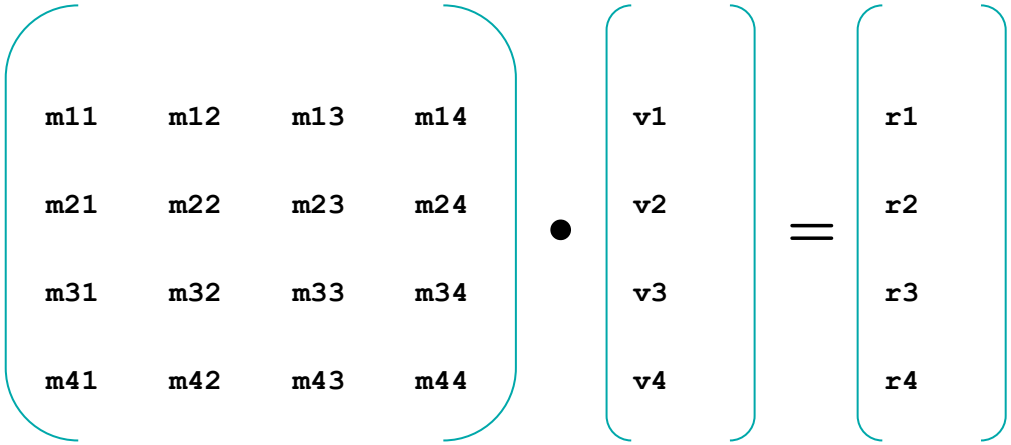
```
// cross (just 3x3)   v3[0] = v1[1]*v2[2]-v1[2]*v2[1]; etc
inline __m128 _mm_cross_ps(__m128 v1, __m128 v2) {
  __m128 v3 = _mm_shuffle_ps(v2, v1, _MM_SHUFFLE(3, 0, 2, 2));
  __m128 v4 = _mm_shuffle_ps(v1, v2, _MM_SHUFFLE(3, 1, 0, 1));

  __m128 v5 = _mm_mul_ps(v3, v4);

  v3 = _mm_shuffle_ps(v1, v2, _MM_SHUFFLE(3, 0, 2, 2));
  v4 = _mm_shuffle_ps(v2, v1, _MM_SHUFFLE(3, 1, 0, 1));

  v3 = _mm_mul_ps(v3, v4);
  const  __m128 neg = _mm_set_ps(0.0f,0.0f,-0.0f,0.0f);
  return _mm_xor_ps(_mm_sub_ps(v5, v3), neg);
}
```

matrix-vector multiplication (SSE2)




```
__m128 t0 = _mm_set1_ps(v[0]);
__m128 t1 = _mm_set1_ps(v[1]);
__m128 t2 = _mm_set1_ps(v[2]);
__m128 t3 = _mm_set1_ps(v[3]);

t0 = _mm_mul_ps(m0, t0);
t1 = _mm_mul_ps(m1, t1);
t2 = _mm_mul_ps(m2, t2);
t3 = _mm_mul_ps(m3, t3);

__m128 r = _mm_add_ps(t0,
           _mm_add_ps(t1,
           _mm_add_ps(t2, t3)));
```

Standard Procedure

```
r1 = v1*m11+v2*m12+v3*m13+v4*m14
r2 = v1*m21+v2*m22+v3*m23+v4*m24
r3 = v1*m31+v2*m32+v3*m33+v4*m34
r4 = v1*m41+v2*m42+v3*m43+v4*m44
```

16 multiplications + 12 additions

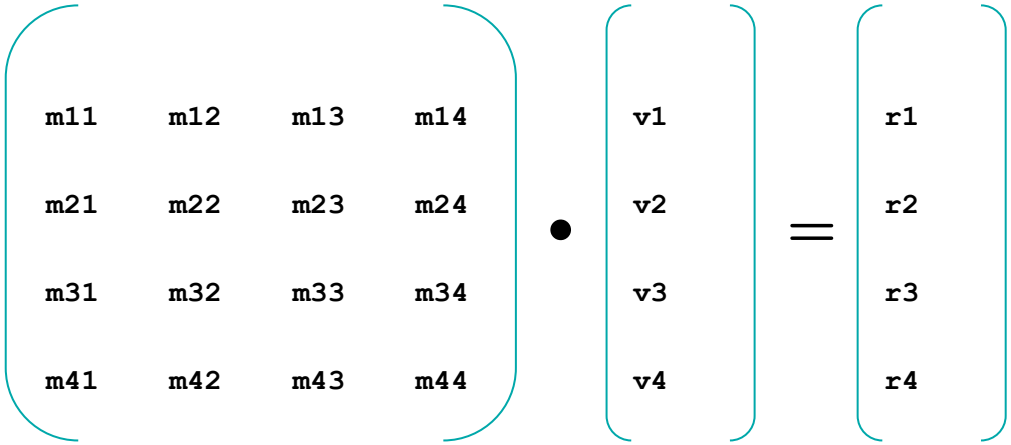
Vector Procedure

```
t1 = [m11, m21, m31, m41] * v1
t2 = [m12, m22, m32, m42] * v2
t3 = [m13, m23, m33, m43] * v3
t4 = [m14, m24, m34, m44] * v4
R = t1 + t2 + t3 + t4
```

4 multiplications + 3 additions

Observed speed-up in this example: ~2x (instead of 4x) using floats

matrix-vector multiplication (SSE4)




```
__m128 t0;
__m128 t1;
__m128 t2;
__m128 t3;

t0 = _mm_dp_ps(m0, v, 0xf1);
t1 = _mm_dp_ps(m1, v, 0xf2);
t2 = _mm_dp_ps(m2, v, 0xf4);
t3 = _mm_dp_ps(m3, v, 0xf8);

__m128 rs = _mm_or_ps(t0,
            _mm_or_ps(t1,
            _mm_or_ps(t2, t3)));
```

Standard Procedure

```
r1 = v1*m11+v2*m12+v3*m13+v4*m14
r2 = v1*m21+v2*m22+v3*m23+v4*m24
r3 = v1*m31+v2*m32+v3*m33+v4*m34
r4 = v1*m41+v2*m42+v3*m43+v4*m44
```

16 multiplications + 12 additions

Vector Procedure

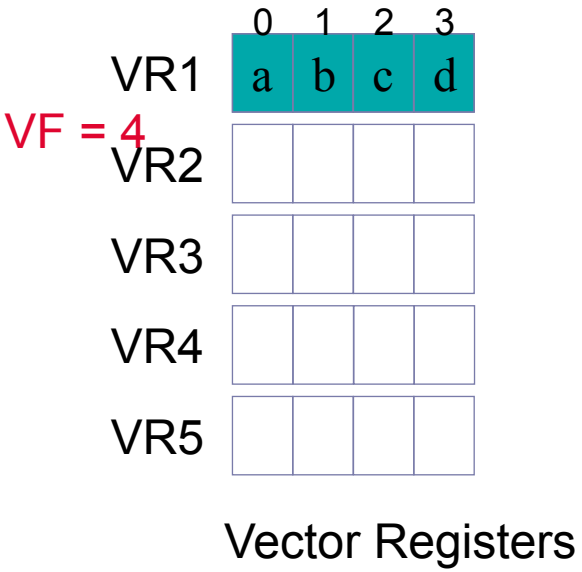
```
t1 = [m11, m12, m13, m14] • v
t2 = [m21, m22, m23, m24] • v
t3 = [m31, m32, m33, m34] • v
t4 = [m41, m42, m43, m44] • v
R = t1 | t2 | t3 | t4
```

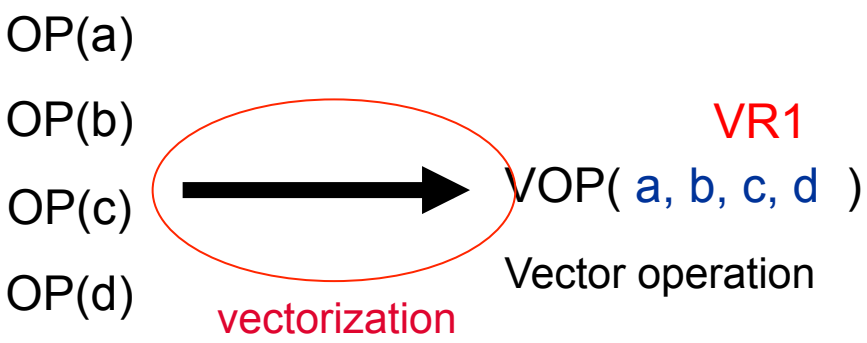
4 dots + 3 ors

Observed speed-up in this example: ~?x (instead of 4x) using floats

SIMD: LOOP VECTORIZATION

What is vectorization





- Data elements packed into vectors
- Vector length → Vectorization Factor (VF)

Data in Memory:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p											

Vectorization

- 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

Intel® compiler Automatically generate SSE instructions for C code.

- Using Intel compiler on the following C code generates SIMD code automatically

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

```
$B4$21:
      movups    xmm1, XMMWORD PTR [rdx+r10*4]
      movups    xmm0, XMMWORD PTR [r8+r10*4]
      mulps     xmm1, xmm0
      movaps    XMMWORD PTR [rcx+r10*4], xmm1
     add       r10, 4
      cmp       r10, rbp
      jl       $B4$21
```

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
```

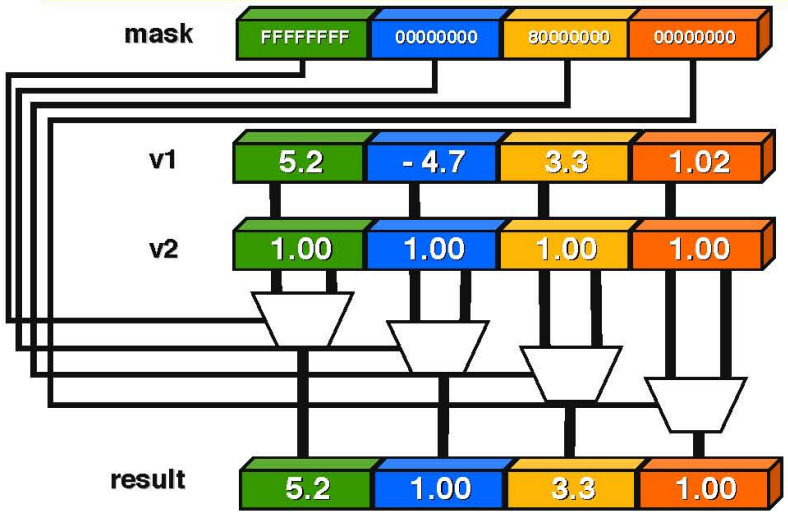
```
for (i=0; i<N; i++){
      B[i] = A[i] + Y
      A[i+1] = B[i] + X
}
```

```
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
```

Blends: To Boost Conditionals SIMD flows

```
/*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);
}
```

A REAL loop

```
void compChi2Scalar(V10 const & ampl, V10 const & err2, Scalar t,
                                Scalar sumAA, Scalar& chi2, Scalar& amp) {
  Scalar sumAf = 0;
  Scalar sumff = 0;
  Scalar const eps = Scalar(1e-6);
  Scalar const 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.arr[it]*(f*err2.arr[it]);
      sumff += f*(f*err2.arr[it]);
    }
  }

  chi2 = sumAA;
  amp = 0;
  if( sumff > 0 ){
    amp = sumAf/sumff;
    chi2 = sumAA - sumAf*amp;
  }
  chi2 *=denom;
}
```

A REAL loop vectorized (1/2)

```
void compChi2(V10 const & ampl, V10 const & err2, Scalar t,
                    Scalar sumAA, Scalar& chi2, Scalar& amp) {
 typedef  V10::Vec Vec;
 Scalar const denom =  Scalar(1)/Scalar(SIZE);

 Vec one = _mm_set1_ps(1); Vec eps = _mm_set1_ps(1.e-6f);
 Vec tv = _mm_set1_ps(t);

 V10 index;
 for(unsigned int it = 0; it != arrsize; ++it)
   index.arr[it]=it;

Vec sumAf;   Vec sumff;
 for(unsigned int it = 0; it < ssesize; it++){
   Vec offset = (index[it]-tv)*overab;
   Vec term1 =  one+offset;
   Vec cmp = cmpgt(term1,eps);

   Vec f =  mathSSE::exp(alpha*(mathSSE::log(term1)-offset) );
   f = cmp&f;
   Vec fe = f*err2[it];
   sumAf = sumAf + V10::Traits::mask(ampl[it]*fe,it);
   sumff = sumff + V10::Traits::mask(f*fe,it);
 }
```

```
 Vec sum = hadd(sumAf,sumff);
 sum = hadd(sum,sum);

 Scalar af = sum[0];
 Scalar ff = sum[1];

 chi2 = sumAA;
 amp = 0;
 if( ff > 0 ){
   amp = af/ff;
   chi2 = sumAA - af*amp;
 }
 chi2 *=denom;
}
```

A REAL loop vectorized (2/2)

```
template<size_t S>
 struct ArrayTraits<float, S> {
   typedef float Scalar;
   typedef Vec4<float> Vec;
   static const size_t size = S;
   static const size_t ssesize = (S+3)/4;
   static const size_t arrsize = 4*ssesize;
   static inline Vec maskLast() { return ArrayMask<Scalar,arrsize-size>::value(); }
   static inline Vec __attribute__((__always_inline__)) mask(Vec v, size_t i) {
     return (i==ssesize-1) ? maskLast()&v : v;
    }
 };

template<typename T, size_t S>
 union Array {
   typedef ArrayTraits<T,S> Traits;
   typedef typename Traits::Vec Vec;
   typename Vec::nativeType vec[Traits::ssesize];
   T __attribute__ ((aligned(16))) arr[Traits::arrsize];

   Vec operator[]( size_t i) { return vec[i];}
   Vec const & operator[]( size_t i) const{ return reinterpret_cast<Vec const &>(vec[i]);}

 };
```

Non-unit Stride Operations

```
_mm_insert_ps(__m128 dst, __m128 src, const int ndx);
_mm_insert_{epi8,epi32,epi64} (__m128i dst, int src, const int ndx);
_mm_extract_ps(__m128 src, const int ndx);
 _mm_extract_{epi8, epi32, epi64} (__m128i src, const int ndx);
```

Strided Load

```
xm1 = _mm_load_ss (a);
xm1 = _mm_insert_ps (xm1, a+stride, 0x10);
xm1 = _mm_insert_ps (xm1, a+2*stride, 0x20);
xm1 = _mm_insert_ps (xm1, a+3*stride, 0x30);
```

Strided Store

```
*a = _mm_extract_ps (x1, 0);
*(a +stride) = _mm_extract_ps (x1, 1);
*(a +2*stride) = _mm_extract_ps (x1, 2);
*(a +3*stride) = _mm_extract_ps (x1, 3);
```

Gather

```
i = _mm_extract_epi32 (xm1, 0);
xm2 = _mm_load_ss (&arr[i]);
i = _mm_extract_epi32 (xm1, 1);
xm2 = _mm_insert_ps (xm2, &arr[i], 0x10);
i = _mm_extract_epi32 (xm1, 2);
xm2 = _mm_insert_ps (xm2, &arr[i], 0x20);
i = _mm_extract_epi32 (xm1, 3);
xm2 = _mm_insert_ps (xm2, &arr[i], 0x30);
```

Scatter

```
i = _mm_extract_epi32 (xm1, 0);
_mm_store_ss (&arr[i], xm2);
i = _mm_extract_epi32 (xm1, 1);
arr[i] = _mm_extract_ps (xm2, 1);
i = _mm_extract_epi32 (xm1, 2);
arr[i] = _mm_extract_ps (xm2, 2);
i = _mm_extract_epi32 (xm1, 3);
arr[i] = _mm_insert_ps (xm2, 3);
```

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

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);
```

What to retain

- Vectorization boosts performance w/o loss of accuracy
- “Horizontal vectors” have perfect fit to geometry and physics
- Loop vectorization requires algorithm to be “streamlined”
  - in many cases algorithms and code need to be re-engineered, even redesigned, to obtain maximal efficiency

What to bring back home

- 4D Vector classes
- Vectorized math (cephes)
- Vectorized random generator
- Helper class for loop vectorization