Table of Contents

Fast and precise Sum of Logs

Sums of many (millions) of logarithms are very common in statistics, in particular for the computation of LogLikehoods. Here we what to show how, exploiting the IEEE754 representation of floating point numbers one can perform a sum of a large numbers of logarithms in fast and precise way.

Sum of `std::log`

Our baseline is the standard way to sum logs

```
double s=0.;
for (int i=0;i!=NN;++i)
   s+= std::log(double(r[i]));
```

Actually as baseline for precision we will accumulate the sum in a `__float128`.

Sum of `vdt::log`

In alternative we can use a faster, vectorizable implementations such as vdt

```
double s=0.;
for (int i=0;i!=NN;++i)
   s+= vdt::fast_log(double(r[i]));
```

using `frexp`

At this point we wish to exploit the fact that IEEE754 representation of floating point numbers is a *mantissa*, in the form `1.m` and an *exponent* in base `2`. So the sum of logs (base 2) is just the sum of the exponents and the log of the product of the mantissas. exponent and mantissa can be obtained by the library function `frexp`.

```
int sf=0; double mf=1;
for (int i=0;i!=NN; ++i) {
  int er=0; double mr = ::frexp(r[i],&er);
  sf+=er; mf*=mr;
  mf = ::frexp(mf,&er); sf+=er;
}
double s=sf+std::log2(mf);
```

optimizing the use `frexp`

the second `frexp` is required to avoid the product to underflow (the very same reason one perform a sum of logs!) The mantissa returned by `frexp` is a number in the interval `[0.5,1]`. For double precision numbers we can easily multiply 512 of them w/o risk of underflow. This essentially reduces by a factor 2 the number of calls to `frexp`.

```
int sf=0; double mf=1.;
for (int i=0;i<NN; i+=512) {
  double mi=1.f;
  for (auto k=i; k!=std::min(i+512,NN); ++k) {
    int er=0; double mr = ::frexp(r[k],&er);
    sf+=er; mi*=mr;
  }
  int ei=0; mf = ::frexp(mf*mi,&ei); sf+=ei;
}
double s=sf+std::log2(mf);
```

inlining and vectorizing `frexp`

As for `log` we can easily provide a version of `frexp` that will inline and vectorize

```
  union binary64 {
    binary64() : ui64(0) {};
    binary64(double ff) : f(ff) {};
    binary64(int64_t ii) : i64(ii){}
    binary64(uint64_t ui) : ui64(ui){}

    uint64_t ui64; /* unsigned int */
    int64_t i64; /* Signed int */
    double f;
  };

inline
void frex(double x, int & er, double & mr) {

  binary64 xx,m;
  xx.f = x;

  // as many integer computations as possible, most are 1-cycle only, and lots of ILP.
  int e=  int( ( xx.ui64 >> 52) & 0x7FF) -1023; // extract exponent
  m.ui64 = (xx.ui64 & 0x800FFFFFFFFFFFFFULL) | 0x3FF0000000000000ULL; // extract mantissa as an F

  long long adjust = (xx.ui64>>51)&1; // first bit of the mantissa, tells us if 1.m > 1.5
  m.i64 -= adjust << 52; // if so, divide 1.m by 2 (exact operation, no rounding)
  e += adjust;           // and update exponent so we still have x=2^E*y

  er = e;
  // now back to floating-point
  mr = m.f; //
  // all the computations so far were free of rounding errors...
}
```

which returns the mantissa in the interval `[.75,1.5]`

we can use this to sum the logs as above

```
int sf=0; double mf=1.;
for (int i=0;i<NN; i+=512) {
  double mi=1.f;
  for (auto k=i; k!=std::min(i+512,NN); ++k) {
     int er=0; float mr=0;  frex(r[k],er,mr);
     sf+=er; mi*=mr;
  }
  int ei=0; frex(mf*mi,ei,mf); sf+=ei;
}
double s=sf+std::log2(mf);
```

In this case the double loop is required also for vectorization purposes as the `frex` of `mf`, the reduction product, will not vectorize being not associative as implemented.

using `floats`

We shall note that the large sum is accumulated in the integer. Double precision is required above only for precision issues not for enlarging the dynamic range. If precision in not an issue floating points can be used to speed up the computation above

```
  union binary32 {
    binary32() : ui32(0) {};
    binary32(float ff) : f(ff) {};
```

```
    binary32(int32_t ii) : i32(ii){}
    binary32(uint32_t ui) : ui32(ui){}

    uint32_t ui32; /* unsigned int */
    int32_t i32; /* Signed int */
    float f;
  };

inline
void frex(float x, int & er, float & mr) {

  binary32 xx,m;
  xx.f = x;

  // as many integer computations as possible, most are 1-cycle only, and lots of ILP.
  int e= (((xx.i32) >> 23) & 0xFF) -127; // extract exponent
  m.i32 = (xx.i32 & 0x007FFFFF) | 0x3F800000; // extract mantissa as an FP number

  int adjust = (xx.i32>>22)&1; // first bit of the mantissa, tells us if 1.m > 1.5
  m.i32 -= adjust << 23; // if so, divide 1.m by 2 (exact operation, no rounding)
  e += adjust;           // and update exponent so we still have x=2^E*y

  er = e;
  // now back to floating-point
  mr = m.f; //
  // all the computations so far were free of rounding errors...
}

int sf=0; float mf=1.f;
for (int i=0;i<NN; i+=128) {
   float mi=1.f;
   for (auto k=i; k!=std::min(i+128,NN); ++k) {
      int er=0; float mr=0; frex(r[k],er,mr);
      sf+=er; mi*=mr;
   }
   int ei=0; frex(mf*mi,ei,mf); sf+=ei;
}
float s=sf+std::log2(mf);
```

using only integers

The attentive reader should have noticed that the mantissa is actually represented by a "fixed point number" of the from `1.m` where `m` has 23 bits for single precision and 52 for double precision. So one can perform the computation above using "fixed point algebra" (actually just a fixed point multiplication is involved)

We shall therefore extract the mantissa as integer and then multiply two "fixed point numbers"

```
inline
void irex(float x, int & er, unsigned int & mr) {

  binary32 xx;
  xx.f = x;

  er = (((xx.ui32) >> 23) & 0xFF) -127; // extract exponent
  mr = (xx.ui32 & 0x007FFFFF) | 0x00800000; // extract mantissa as an integer number

}

inline
unsigned int mult(unsigned int a, unsigned int b) {
  constexpr unsigned int Q = 23;
  constexpr unsigned long long K  = (1 << (Q-1));
  unsigned long long  temp = (unsigned long long)(a) * (unsigned long long)(b); // result type is
  // Rounding; mid values are rounded up
```

```
  temp += K;
  // Correct by dividing by base
  return (temp >> Q);
}
```

we use this to evaluate the sum of logs taking care, as usual, not to overflow the product of mantissas

```
int si=0;
unsigned int pi=0.; unsigned int ipi=0;
irex(1.f,si,ipi); pi=ipi;
for (int i=0;i!=NN;++i) {
  int er=0; unsigned int mr=0; irex(r[i],er,mr);
  si+=er; pi=mult(pi,mr);
  //  if (pi >= 0x80000000)  { pi/=2; si++;} // avoid overflow
  unsigned int ov = pi >> 31;  pi>>=ov; si+=ov; // avoid overflow
}
float s = si+std::log2(pi)-23;
```

optimize the use of integer multiplication

The loop above will not vectorize because the multiplication of the mantissas is not associative as implemented. One can try to vectorize by hand. What the the compiler does is actually to *unroll* the inner loop, which speed up the computation by 20%

```
int si=0; unsigned int ipi=0;
irex(1.f,si,ipi);
unsigned int pi[4]={ipi,ipi,ipi,ipi};
int sk[4]={si,si,si,si};
for (int i=0;i<NN;i+=4) {
  for (int k=0; k!=4; ++k) {
    int er=0; unsigned int mr=0; irex(r[i+k],er,mr);
    sk[k]+=er; pi[k]=mult(pi[k],mr);
    unsigned int ov = pi[k] >> 31;  pi[k]>>=ov; sk[k]+=ov; // avoid overflow
   }
}
si = sk[0]+sk[1]+sk[2]+sk[3]+8+16;
ipi = mult(mult(pi[0]>>4,pi[1]>>4)>>4,mult(pi[2]>>4,pi[3]>>4)>>4);
float s = si+std::log2(ipi)-23;
```

manual vectorization of the use of integer multiplication (SSE)

We can try to overcome the non-associative nature of the C++ implementation of the integer multiplication by manually vectorizing it using gcc vector extensions and the intrinsics corresponding to the `PMULUDQ` instruction (built for this very purpose!)

for SSE (128 bit vectors) the extraction of the mantissa as integer and the multiplication of two "fixed point numbers" will look like

```
typedef unsigned long long __attribute__( ( vector_size( 16 ) ) ) uint64x2_t;
typedef signed long long __attribute__( ( vector_size( 16 ) ) ) int64x2_t;
typedef signed int __attribute__( ( vector_size( 16 ) ) ) int32x4_t;
typedef unsigned int __attribute__( ( vector_size( 16 ) ) ) uint32x4_t;
typedef float __attribute__( ( vector_size( 16 ) ) ) float32x4_t;

  union binary128 {
    binary128() : ul{0,0} {};
    binary128(float32x4_t ff) : f(ff) {};
    binary128(int32x4_t ii) : i(ii){}
    binary128(uint32x4_t ii) : ui(ii){}
    binary128(int64x2_t ii) : l(ii){}
    binary128(uint64x2_t ii) : ul(ii){}
```

```
    __m128i i128;
    float32x4_t f;
    int32x4_t i;
    uint32x4_t ui;
    int64x2_t l;
    uint64x2_t ul;
  };

inline
void irex(float32x4_t x, int32x4_t & er, uint32x4_t & mr) {

  binary128 xx;
  xx.f = x;

  // as many integer computations as possible, most are 1-cycle only, and lots of ILP.
  er = (((xx.i) >> 23) & 0xFF) -127; // extract exponent
  mr = (xx.ui & 0x007FFFFF) | 0x00800000; // extract mantissa as an integer number

}

inline
uint32x4_t mult(uint32x4_t a, uint32x4_t b) {

  binary128 temp1, temp2;
  constexpr int Q = 23;
  constexpr unsigned long long K  = (1 << (Q-1));
  temp1.i128 = _mm_mul_epu32(__m128i(a),__m128i(b));
  temp1.ul += K;  temp1.ul >>= Q;  // results are in position 0 and 2

  a.ul >>= 32; b.ul >>= 32;  // move "odd" integers in position
  /*  same speed on SB!
  constexpr int32x4_t mask{1,0,3,2};
  a.ui = __builtin_shuffle(a.ui,mask);
  b.ui = __builtin_shuffle(b.ui,mask);
  */

  temp2.i128 = _mm_mul_epu32(__m128i(a),__m128i(b));
  temp2.ul += K;  temp2.ul >>= Q;

  temp2.ul <<= 32;  // results are now in position 1 and 3
  temp1.ul |=temp2.ul;

  /*
  constexpr int32x4_t mask2{0,4,2,6};
  temp.i = __builtin_shuffle(temp1.i,temp2.i,mask2);
  */
  return temp1.ui;
}
```

and the loop will become

```
int si=0; unsigned int ipi=0;
irex(1.f,si,ipi);
uint32x4_t pi = {ipi,ipi,ipi,ipi};
int32x4_t sk = {si,si,si,si};
for (int i=0;i<NN;i+=4) {
  int32x4_t er; uint32x4_t mi;
  float32x4_t ri{r[i+0],r[i+1],r[i+2],r[i+3]};
  irex(ri,er,mi);  sk+=er;
  pi = mult(pi,mi);
  binary128 ov(pi >> 31);  pi>>=ov.ui; sk+=ov.i; // avoid overflow
}
si = sk[0]+sk[1]+sk[2]+sk[3]+8+16;
ipi = mult(mult(pi[0]>>4,pi[1]>>4)>>4,mult(pi[2]>>4,pi[3]>>4)>>4);
float s = si+std::log2(ipi)-23;
```

optimization of the vectorization

One can gain some speed keeping the "mantissas" as `unsigned long long`, This makes the `mult` function a bit *ad-hoc*, still the gain in not negligible.

So one rewrites the multiplication as

```
inline
void mult(uint64x2_t & aa1, uint64x2_t & aa2, uint32x4_t bb) {
  using namespace approx_math;
  binary128 a1(aa1), a2(aa2), b(bb);
  constexpr int Q = 23;
  constexpr unsigned long long K  = (1 << (Q-1));
  a1.i128 = _mm_mul_epu32(a1.i128,b.i128);
  a1.ul += K;  a1.ul >>= Q;

  b.ul >>= 32;  // move "odd" integers in position

  a2.i128 = _mm_mul_epu32(a2.i128,b.i128);
  a2.ul += K; a2.ul >>= Q;
  aa1 = a1.ul; aa2=a2.ul;
}
```

and the loop will become

```
int si=0; unsigned int ipi=0;
irex(1.f,si,ipi);
uint64x2_t p1 = {ipi,ipi};
uint64x2_t p2 = {ipi,ipi};
uint64x2_t s1{0,0};
int32x4_t sk = {si,si,si,si};
for (int i=0;i<NN;i+=8) {
  for (int j = i; j<i+8; j+=4)
  {
     int32x4_t er; uint32x4_t mi;
     float32x4_t ri{r[j+0],r[j+1],r[j+2],r[j+3]};
     irex(ri,er,mi);  sk+=er;

     mult(p1,p2,mi);
     binary128 ov1(p1 >> 31);  p1>>=ov1.ul; s1+=ov1.ul; // avoid overflow
     binary128 ov2(p2 >> 31);  p2>>=ov2.ul; s1+=ov2.ul; // avoid overflow
   }
}
si = sk[0]+sk[1]+sk[2]+sk[3]+8+16; si+=s1[0]+s1[1];
ipi = mult(mult(p1[0]>>4,p1[1]>>4)>>4,mult(p2[0]>>4,p2[1]>>4)>>4);
float s = si+std::log2(ipi)-23;
```

the manual unroll seems to gain few percent more .

AVX2

On INTEL HASHWELL machine we can use AVX2 instruction set that extends the integer vector to 256 bits. the above code will therefore become

```
 union binary256 {
    binary256() : ul{0,0} {};
    binary256(float32x8_t ff) : f(ff) {};
    binary256(int32x8_t ii) : i(ii){}
    binary256(uint32x8_t ii) : ui(ii){}
    binary256(int64x4_t ii) : l(ii){}
    binary256(uint64x4_t ii) : ul(ii){}
```

```
    __m256i i256;
    float32x8_t f;
    int32x8_t i;
    uint32x8_t ui;
    int64x4_t l;
    uint64x4_t ul;
  };

inline
void irex(float32x8_t x, int32x8_t & er, uint32x8_t & mr) {
  using namespace approx_math;

  binary256 xx;
  xx.f = x;

  // as many integer computations as possible, most are 1-cycle only, and lots of ILP.
  er = (((xx.i) >> 23) & 0xFF) -127; // extract exponent
  mr = (xx.ui & 0x007FFFFF) | 0x00800000; // extract mantissa as an integer number

}

inline
void multI2(uint64x4_t & aa1, uint64x4_t & aa2, uint32x8_t bb) {
  using namespace approx_math;
  binary256 a1(aa1), a2(aa2), b(bb);
  constexpr int Q = 23;
  constexpr unsigned long long K  = (1 << (Q-1));
  a1.i256 = _mm256_mul_epu32(a1.i256,b.i256);
  a1.ul += K;  a1.ul >>= Q;

  b.ul >>= 32;  // move "odd" integers in "even" positions

  a2.i256 = _mm256_mul_epu32(a2.i256,b.i256);
  a2.ul += K; a2.ul >>= Q;
  aa1 = a1.ul; aa2=a2.ul;
}

    {
      // full integer ( avx2 vectorized???)
      t23 -= rdtsc();
      int si=0; unsigned int ipi=0;
      irex(1.f,si,ipi);
      // uint32x4_t pi = {ipi,ipi,ipi,ipi};
      uint64x4_t p1 = {ipi,ipi,ipi,ipi};
      uint64x4_t p2 = {ipi,ipi,ipi,ipi};
      uint64x4_t s1{0,0,0,0};
      int32x8_t sk = {si,si,si,si,si,si,si,si};
      for (int i=0;i<NN;i+=16) {
   for (int j = i; j<i+16; j+=8)
   {
     int32x8_t er; uint32x8_t mi;
     float32x8_t ri{r[j+0],r[j+1],r[j+2],r[j+3],r[j+4],r[j+5],r[j+6],r[j+7]};
     irex(ri,er,mi);  sk+=er;
     multI2(p1,p2,mi);
     approx_math::binary256 ov1(p1 >> 31);  p1>>=ov1.ul; s1+=ov1.ul; // avoid overflow
     approx_math::binary256 ov2(p2 >> 31);  p2>>=ov2.ul; s1+=ov2.ul; // avoid overflow
   }
      }
      si = sk[0]+sk[1]+sk[2]+sk[3]+sk[4]+sk[5]+sk[6]+sk[7] + 14*4; si+=s1[0]+s1[1]+s1[2]+s1[3];
      ipi = mult(
       mult(mult(p1[0]>>4,p1[1]>>4)>>4,mult(p2[0]>>4,p2[1]>>4)>>4)>>4,
       mult(mult(p1[2]>>4,p1[3]>>4)>>4,mult(p2[2]>>4,p2[3]>>4)>>4)>>4
        );


      s= si + std::log2(ipi)-23;
```

integer "extended precision"

we can also code the integer only code for "double precision" using `int128` math

```
inline
void irex(double x, int & er, unsigned long long & mr) {
  using namespace approx_math;

  binary64 xx;
  xx.f = x;

  // as many integer computations as possible, most are 1-cycle only, and lots of ILP.
  er =  int( ( xx.ui64 >> 52) & 0x7FF) -1023; // extract exponent
  mr = (xx.ui64 & 0x000FFFFFFFFFFFFFULL) | 0x0010000000000000ULL; // extract mantissa as an integ

}

inline
__uint128_t multD(__uint128_t a, __uint128_t b) {
  constexpr int Q = 52;
  constexpr unsigned long long K  = (1UL << (Q-1));
  auto temp = a*b;
  temp += K;
  return temp >> Q;
}
```

with the unrolled loop becoming

```
int si=0; unsigned long long ipi=0;
irex(1.,si,ipi);
__uint128_t pi[4]={ipi,ipi,ipi,ipi};
int sk[4]={si,si,si,si};
for (int i=0;i<NN;i+=4) {
    for (int k=0; k!=4; ++k) {
    int er=0; unsigned long long mr=0;
    irex(double(r[i+k]),er,mr);
    sk[k]+=er; pi[k]=multD(pi[k],mr);
    unsigned long long ov = pi[k] >> 63;  pi[k]>>=ov; sk[k]+=ov; // avoid overflow
  }
}
constexpr int shift=8;
si = sk[0]+sk[1]+sk[2]+sk[3]+ 6*shift;
ipi = multD(multD(pi[0]>>shift,pi[1]>>shift)>>shift,multD(pi[2]>>shift,pi[3]>>shift)>>shift);
double s = si+std::log2(ipi)-52;
```

It is interesting to note how the result is identical to the last bit to the "frexp double precision version", only 7 times slower.

performance

To measure the performance of these various methods we have "summed the log" of one millions random numbers in the interval `[0,1]`. to further verify the precision we have also used one millions random numbers in the intervals `[0,0.1]` and `[0.9,1]`

the speed is measured using **`rdtscp`** on an INTEL SandyBridge CPU. we used gcc version 4.9. the value reported is normalized to the number of log summed (one million). As we are reading 4MB bandwidth to L3 cache matters. We therefore time also the loop taking the simple sum as float. the whole benchmark is run 102 times. The precision is the maximal relative deviation observed for the total sum from the result using `std::log2` including the two intervals `[0,0.1]` and `[0.9,1]`.

	SandyBridge AVX		Hashwell AVX2	
algorithm	speed	precision	speed	precision
sum	0.42		0.42	
float128	197	0	175	0
std	68.5	1e-13	55.4	1e-13
vdt	16.9	3e-9	8.6	3e-9
frexp	31.2	0	26.5	0
opt frexp	15.2	0	12.0	0
vec rexp d	2.07	0	1.10	0
vec rexp s	1.63	4e-8	1.14	4e-8
integer	7.8	3e-9	6.3	3e-9
opt integer	5.8	3e-9	3.8	3e-9
vect integer	6.8	3e-9		
opt vect int	5.6	3e-9	2.7	3e-9
avx2 int			1.47	3e-9
long long	15	0	12.6	0

it should be noted that the results of float128, various versions of frexp and "long long integer" are all identical to last bit.

We also note how on Hashwell also standard see code improves due to the increased ILP

note that on *MAC* the speed of `std` is 25.5.

This topic: LCG > VIFastSumLog
Topic revision: r14 - 2018-07-20 - VincenzoInnocente