The Metalibm project

Florent de Dinechin

Introduction

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour in 2002
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour in 2002
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration: a Ph.D. thesis (2002)

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour in 2002
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration: a Ph.D. thesis (2002)

Conclusion was:

- performance and memory consumption of CR elem function is OK

First function development in Arénaire

First correctly rounded elementary function in CRLibm

- `exp` by David Defour in 2002
- worst-case time $T_2 \approx 10{,}000$ cycles
- complex, hand-written proof
- duration: a Ph.D. thesis (2002)

Conclusion was:

- performance and memory consumption of CR elem function is OK
- problem now is: performance and coffee consumption of the programmer

Latest function developments in Arénaire

C. Lauter at the end of his PhD in 2008

- development time for sinpi, cospi, tanpi:

Latest function developments in Arénaire

C. Lauter at the end of his PhD in 2008

- development time for sinpi, cospi, tanpi: 2 days
- worst-case time $T_2 \approx 1{,}000$ cycles
- Coq-verified proof generated along with the code

... as a result of three more PhDs

- development of **Sollya**:
  the Swiss army knife of polynomial approximation
- development of the **Gappa** high-level proof assistant:
  Gappa is to Coq what C is to assembly language

Summary of the progress made

Performance of a two-step correctly-rounded implementation

$$T_{\text{avg}} = T_1 + p_2 T_2$$

- Reduction of T_1 by learning from Intel
- Reduction of p_2 by automating the computation of tight $\overline{\varepsilon}_1$
  (p_2 is proportional to $\overline{\varepsilon}_1$)
- Reduction of T_2 by computing just right
  (design of an ad-hoc multiple-precision library)
- Reduction of coffee consumption by automating the whole thing

Summary of the progress made

Performance of a two-step correctly-rounded implementation

$$T_{\text{avg}} = T_1 + p_2 T_2$$

- Reduction of T_1 by learning from Intel
- Reduction of p_2 by automating the computation of tight $\overline{\varepsilon}_1$
  (p_2 is proportional to $\overline{\varepsilon}_1$)
- Reduction of T_2 by computing just right
  (design of an ad-hoc multiple-precision library)
- Reduction of coffee consumption by automating the whole thing

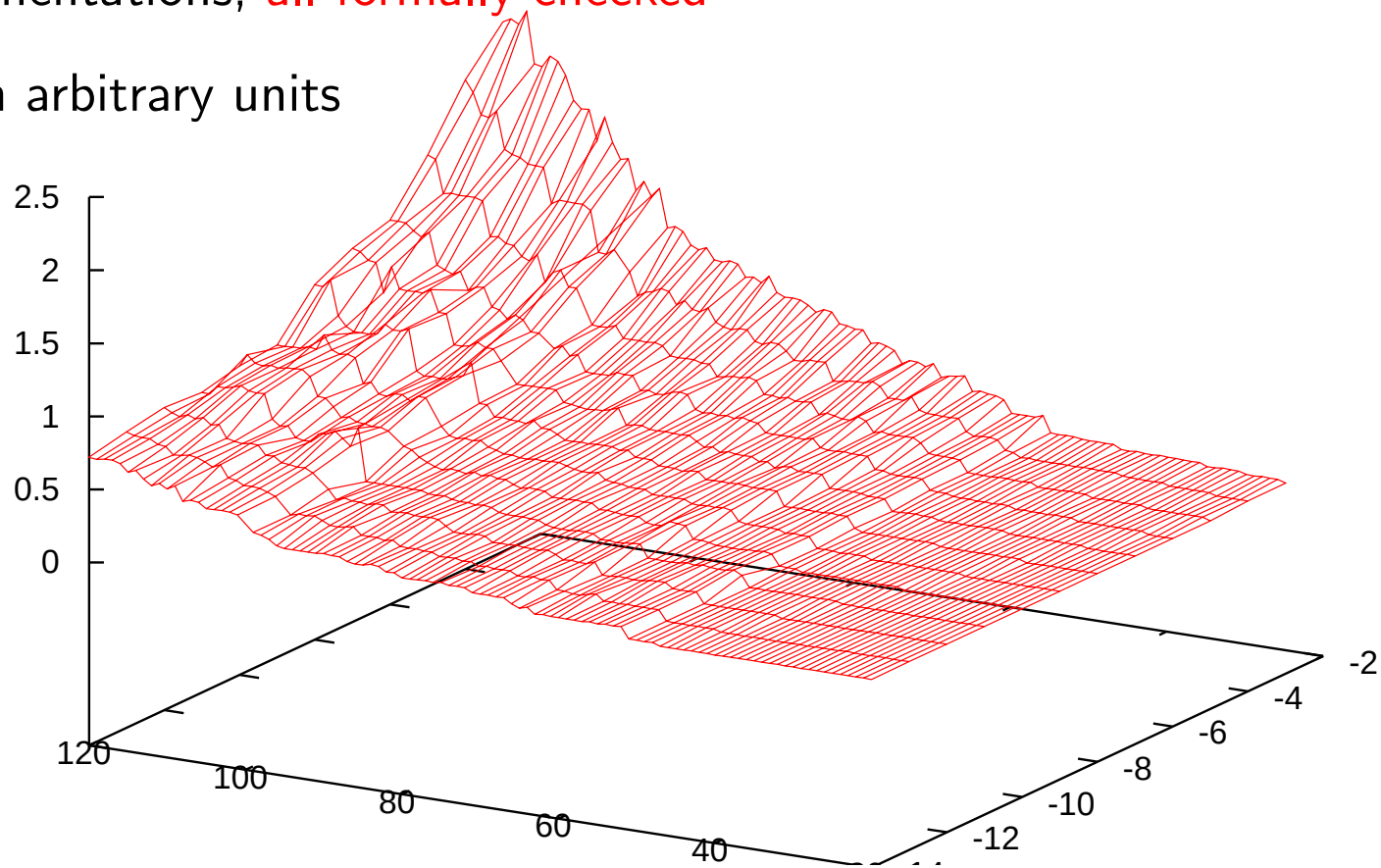
The MetaLibm narrow-minded vision

Automate libm expertise so that a new, correct libm can be written for a new processor/context in minutes instead of months.

Christoph Lauter's metalibm

- Example: $\log(1+x)$
- Two parameters
  - k from 1 to 13, defines table size
  - target accuracy, between 20 and 120 bits
- 1203 implementations, all formally checked

z axis: timings in arbitrary units



The MetaLibm open-ended vision

- We needed to automate the development of code+proof for the C11 elementary functions

The MetaLibm open-ended vision

- We needed to automate the development of code+proof for the C11 elementary functions
- Now that this is (almost) done, we may *open up the set of functions/precisions/performance constraints*

An ANR-funded project

- metalibm/C11
  - focus on performance (match hand-coded libraries)
  - genericity in target processor
  - hand-code what we are unable (yet) to automate:
    range reductions, floating-point trickery, ...
- metalibm/OpenEnded
  - genericity in input
- for processors (x86, Kalray), but also for GPUs and FPGAs
- elementary functions, but also DSP filters

(ANR is the French clone of the NSF)

Filters for digital signal processing?

They are very similar to elementary function:

- Approximation of a transfer function:
  - as a *finite impulse response* filter (FIR)
    (similar to polynomial)
  - as an *infinite impulse response* filter (IIR)
    (similar to a rational fraction)

  The approximation algorithm is a variant of Remez.

- Evaluation: as a sum of product
  - Possible symmetries and other evaluation tricks
  - Rounding error control actually easier for FIR than for polynomials.

Current state of the art

All sorts of cooking recipes that could be better understood and automated.

Stuff that is fully automated

... or at least fully understood:

Polynomial approximation

- Approximation by a polynomial with machine coefficients
- Horner evaluation "just right"
  - each coefficient, each multiplication, each addition have a nominal size
  - relevant for multiple precision, and hardware generation
- many of the classical tricks (odd polynomials, etc) formalized there

Stuff that is almost fully automated

Estrin-like polynomial evaluation

State of the art is CGPE:

- navigation of the space of polynomial evaluation schemes
- taking into account both performance and accuracy
- generation of code + Gappa proof

To embrace and extend.

Stuff that is not yet fully automated

Range reduction

Two points of view here:

- for a given range reduction, automate parameter selection
  - case-by-case
  - relatively easy, if time-consuming (both dev. and proc. time)
  - Metalibm/C11 stuff
- automatically select a range reduction
  - among a known set: relatively easy (based on the previous)
  - Ch. Lauter explores doing it numerically for a black-box function
  - for an open-ended function: very difficult
  - Metalibm/OpenEnded stuff

Open-ended input

- As an arbitrary expression + interval
- As a differential equation

Advertising

Dynamic Dictionary of Mathematical Functions

`http://ddmf.msr-inria.inria.fr/`

A high-level specification that could help us with range reduction?

Beyond the horizon

Functions of several variables

This ANR funding starts in october 2013

We should therefore carefully avoid working on the subject until then.

This ANR funding starts in october 2013

We should therefore carefully avoid working on the subject until then.

However,

- Christoph Lauter in Paris (with a PhD student) has been progressing on a metalibm/OpenEnded prototype written in Sollya. I'm sure he will have an opportunity to demo it himself.

This ANR funding starts in october 2013

We should therefore carefully avoid working on the subject until then.

However,

- Christoph Lauter in Paris (with a PhD student) has been progressing on a metalibm/OpenEnded prototype written in Sollya. I'm sure he will have an opportunity to demo it himself.
- We got a donation from Intel in 2013 that helped bootstrap a metalibm/C11 prototype
  - which has already been forked twice, once by Kalray, once by Intel

This ANR funding starts in october 2013

We should therefore carefully avoid working on the subject until then.

However,

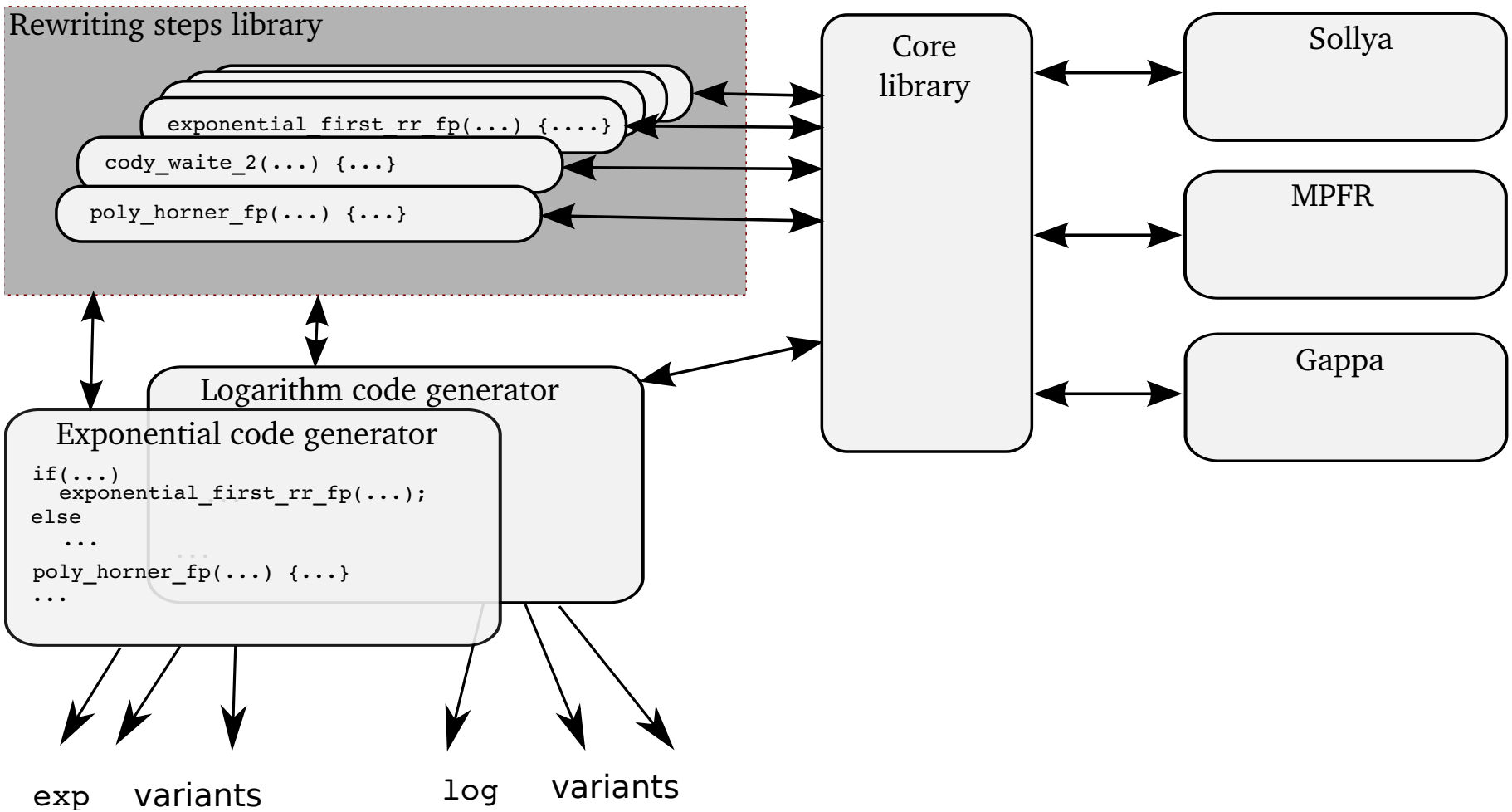
- Christoph Lauter in Paris (with a PhD student) has been progressing on a metalibm/OpenEnded prototype written in Sollya. I'm sure he will have an opportunity to demo it himself.
- We got a donation from Intel in 2013 that helped bootstrap a metalibm/C11 prototype
  - which has already been forked twice, once by Kalray, once by Intel

Already a lot of fragmentation, and the project hasn't started yet...

Metalibm/C11 prototype

An ad-hoc approach

How to develop/retarget functions in lower time?



First, a Sollya interface for Python

Why?

- Python as a scripting language should be good enough
- Focus Sollya development on its core functionalities
  - faithful evaluation of arbitrary expression
  - certified supremum norm
  - machine-efficient polynomial approximation
- Integrate Sollya in Sage (SoSage?)

What?

- A Python module that adds the type SollyaObject to Python
- Autogenerated wrappers for most Sollya functions

Still TODO

- Better typed interfaces, moving away from Sollya's
- PythonSollya should be separated from the metalibm project.

The CFunction class of Metalibm/C11

Main attributes

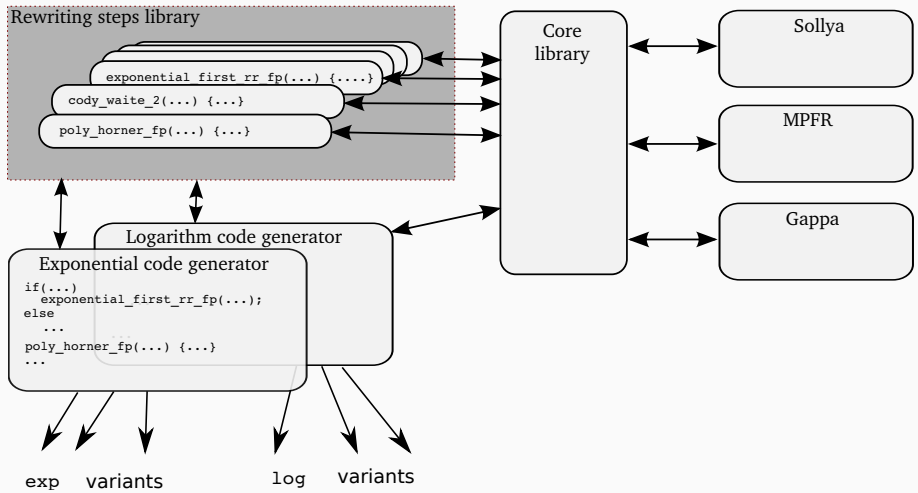
- basename (string)
- io_format (Format)
- input_list
- output_list
- processor (Processor class)
- accuracy (int)
- correct_rounding (boolean)
- manage_subnormals (boolean)
- vectorize (boolean)
- eval_Estrin (boolean)

Main methods

- *gen_code()*, gen_header(), gen_declaration()
- *gen_emulation_code()*
- gen_test_program(), gen_exhaustive_test_program()

A metaexp

Objective: evaluate the "rewriting step" approach



- Set up the core Metalibm/C11 framework
  - a C11Function class
  - a Processor class
  - a Format class (including floating, fixed, and integer)
  - a Polynomial class

In practice

The problem

evaluate e^x faithfully to a double for x a double

First step: invent a range reduction

Here is its ideal mathematical description:

$$\left\{\begin{array}{l} k = \left\lfloor x \times \frac{1}{\ln(2)} \right\rceil \\ \text{and} \\ y = x - k \times \ln(2) \end{array}\right. \Longrightarrow \left\{\begin{array}{l} k \in \mathbb{Z} \\ \text{and} \\ y \in [-\frac{ln(2)}{2}, \frac{ln(2)}{2}] \\ \text{and} \\ e^x = 2^k \times e^y \end{array}\right. \quad (1)$$

Second step: refine to a machine-implementable version

$$k = \left\lfloor x \times \frac{1}{\ln(2)} \right\rceil \tag{2}$$

$$k_1 = \left\lfloor x \times \frac{1}{\ln(2)} + \delta_k \right\rceil, \quad \delta_k \in I_{\delta_k}, \quad k_1 - k \in I_k \tag{3}$$

$$y_1 = x - k_1 \times \ln(2) + \delta_y, \quad y \in I_y, \quad \delta_y \in I_{\delta_y} \tag{4}$$

$$p_1 = e^{y_1} + \delta_p, \quad \delta_p \in I_{\delta_p} \tag{5}$$

$$r = 2^{k_1} \times p_1 \tag{6}$$

Can this two-step derivation be found by a program?

I don't think so.
So I consider (2) to (6) as the starting point of a metaexp.

Meta-skeleton for the exp

$$(I_{\delta_k}, I_k) = \text{genCodeForComputingK}(\text{formatX}, ...) \quad (7)$$

$$(I_{\delta_y}, I_y) = \text{genCodeForComputingY}(I_{\delta_k}, I_k, ...) \quad (8)$$

$$(I_{\delta_p}) = \text{genCodeForPolyApprox}(\text{"}exp(x)\text{"}, targetPrecision, I_y, ...) \quad (9)$$

$$(I_{\delta_r}) = \text{genCodeForReconstruction}(I_k, ...) \quad (10)$$

Actual metaexp skeleton

```
def gen_code(self):
    # Build the code
    self.gen_code_for_k("x")
    self.gen_code_for_y()
    self.gen_code_for_poly()
    self.gen_code_for_reconstruction()
    self.gen_code_for_exceptions()
```

All the previous variables have become global class attributes

- more readable
- but dependencies lost

```
def gen_code_for_k(self, X):
    self.code.declare("k", int32)
    self.code.declare("kf", self.fp_format)
    c=askSollya("1/log(2)")
    roundedc = round(c, self.fp_format.precision, RN)
    self.code.declare_const("invLog2", self.fp_format, roundedc)
    self.code.declare("nrK", self.fp_format)
    self.code <<  "nrK" + " = " +  "invLog2 * " + X +"; /* not rounded K *
    self.processor.genCodeForFloatToInt("k", "kf", "nrK", self.fp_format,

    # Error computation -- at some point to be delegated to Gappa
    # Error of storing roundedc and not log(2)
    delta1 = round(c-roundedc, 24, RU) # minor TODO: double rounding here
    # Error of the floating point multiplication by roundedc
    maxdelta2 = abs(self.fp_format.u*c)
    I_inf= round((-maxdelta2+delta1)*self.max_value_for_finite_output, 24,
    I_sup= round((maxdelta2+delta1)*self.max_value_for_finite_output, 24,
    self.I_deltak = (I_inf, I_sup)

    if (self.I_deltak[0] <= -1)  or (self.I_deltak[1] >= 1):
        raise Exception('I_deltak to large to ensure I_k is {-1,0,1}')
```

more comments in the actual metalibm/metaexp.py

The Processor class

... provides code generation services.

- methods with a failsafe, portable default
- actual processor classes inherit them and may overload them (with whatever intrinsincs etc)
- so the same source is indeed optimized for a range of processor

Current examples:

- `possible_fma`
- `true_fma`
- variants of `float_to_int` (using magic constants, using `nearbyint`, using intrinsics

TODO: capture higher-level capabilities, such as SIMD capabilities.

All this to generate this

```
float invLog2 = 0x1.715476p0f;
float rnd_cst = 12582912.f;

float nrK;
float nrKrounded;
float kf;
int32_t k;

nrK = invLog2 * x; /* not rounded K */
/* float rounded to an int using the magic constant */
nrKrounded = (nrK + rnd_cst) - rnd_cst; /* this rounds to the nearest int
kf = nrKrounded; /* floating-point rounded result */
k = nrKrounded; /* this float to int conversion is a truncation */
```

Perfs for exp (see metalibm/tests/perftests.cc)

- My laptop: Intel(R) Core(TM)2 Duo CPU U9600 @ 1.60GHz
- My desktop: Intel(R) Xeon(TM) CPU E5-1620 0 @ 3.60GHz

Both running XUbuntu 12.10 with gcc 4.7.2

	Core2 U9600	Xeon E5-1620
stock expf	193	45
expf Horner	87	24
expf Estrin	77	27
stock exp	108	60
exp Horner	130	28
exp Estrin	89	36

Disclaimers:

- timings using `__rdtsc()`, usual caveats apply.
- inlining switched on for our code, not for the stock function.

Metaexp: the good

(here I should demo and show code)

- Already 8 useful implementations
- the relevant parts of the error computation properly implemented (float/double, subnormals or not, Estrin or Horner)
  plus the degraded accuracy versions for our physicist friends
- TODO: target ARM (currently only one generic processor)
- TODO: vectorizable version
- TODO: faithful/correct rounding

Metaexp: the bad and the ugly

- No Gappa generation yet
  - bits implemented by Nicolas in the Polynomial class
  - all of metaexp designed for Gappa proof
  - bits out of reach of Gappa: all that manipulates the format
  - these should be properly encapsulated in the Format class, and tested (or proven directly in Coq) there
- Current code doesn't autovectorize with GCC
- Does the structuration as a metaskeleton bring any useful service ? (it was supposed to manage the chaining of errors etc)
  - Nicolas wrote a fixed-point version
  - focused on bringing fixed-point in the framework, not on the "meta" thing.
  - Now need to merge, and answer this important question...

A metalog

Objectives

- A second test of the framework
- Does the structuration as a metaskeleton bring any useful service ?
  - let's try without
- experiment with optimized for latency / optimized for throughput
  - using autovectorisation with gcc 4.7

Perfs for log (see metalibm/tests/perftests.cc)

	Core2 U9600	Xeon E5-1620
stock logf	99	36
logf_horner(opt. for latency)	88	30
the same, autovectorized for SSE2	35	30
logf_horner_v (opt. for throughput)	107	33
the same, autovectorized for SSE2	11	11
stock log	132	86
log_horner (opt. for latency)	171	37

Metalog: what works and what doesn't

- Autovectorization, using standard stable GCC, works for float.
  - Not for double, need to investigate why.
- `-mavx` should bring an improvement on the vectorized code, and doesn't
  - no `%ymmi` in the generated assembly ?!?
  - Either AVX doesn't replicate all SSE2 functions, or GCC is not ready
- I'm not sure I understand how a degree-20 Horner polynomial is evaluated in 37 cycles
- Estrin evaluation would be useful here
  - current implementation not modular enough
  - short-term TODO

A glance at generated code

```
/* Exceptional case filtering, vectorizable */
minfty.ui = 0xff800000; /* minus infinity */
nan.ui = 0x7fc00000; /* nan */
ret_minfty = ((xx.ui & 0x7fffffff) == 0) ? minfty.f : 0.0f; /* x == +/-0 ?
ret_nan = (xx.ui > 0x80000000) ? nan.f : 0.0f;  /* x<0 ?*/
x_is_inf_or_nan = ((xx.ui &  0x7fffffff) >= 0x7f800000) ? xx.f : 0.0f;  /*
exn = ret_minfty + ret_nan + x_is_inf_or_nan;   /* 0.0 if normal or subnor
/* Now remains to add exn somewhere where it will propagate to the result
x_subnormal = (xx.ui < 0x00800000) && (xx.ui > 0);
subnormal_scale = x_subnormal ? 0x1.p48f : 1.0f; /* scale mantissa*/
e_x = x_subnormal ? -127-48 : -127; /* ... and initialize exponent*/
xx.f *= subnormal_scale;
/* Now decompose x into fraction and exponent */
e_x += ((xx.i) >> 23) & ((1<<8)-1); /* extract exponent*/
fraction.i = (xx.i & 0x007fffff); /* extract fraction bits*/
adjust = (fraction.i>>22); /* first non-implicit bit of the fraction, tell
fraction.i = fraction.i |0x3f800000; /* add the exponent of one */
fraction.i -= adjust << 23; /* if m>1.5, divide fraction by 2 (exact opera
e_x += adjust;     /* and update exponent so we still have x = 2^e_x * fra
```

```
/* Now back to floating-point */
y = fraction.f - 1.0f; /* Sterbenz-exact; may cancel but we don't care  */
y += exn; /* exn is either 0.0, or an inf or NaN that will propagate to th
/* Now y in [-0.25, 0.5], and we must evaluate log(1+y) */
/* Horner evaluation */
y2 = y*y;
p9 = c9;
p8 = c8 + y*p9;
p7 = c7 + y*p8;
p6 = c6 + y*p7;
p5 = c5 + y*p6;
p4 = c4 + y*p5;
p3 = c3 + y*p4;
p2 = c2 + y*p3;
p = y + y2*p2;
r = e_x*log_2 + p;
return r;
```

Horner autovectorized to SSE2

thanks to `gcc -O3 -msse2 -finline-limit=1000 -S`

Without

```
mulss %xmm2, %xmm1
subss %xmm10, %xmm0
mulss %xmm2, %xmm0
addss %xmm9, %xmm0
mulss %xmm2, %xmm0
subss %xmm8, %xmm0
mulss %xmm2, %xmm0
addss %xmm7, %xmm0
mulss %xmm2, %xmm0
subss %xmm6, %xmm0
mulss %xmm2, %xmm0
addss %xmm5, %xmm0
mulss %xmm2, %xmm0
subss %xmm4, %xmm0
```

With

```
mulps %xmm1, %xmm2
subps .LC50(%rip), %xmm0
mulps %xmm1, %xmm0
addps .LC51(%rip), %xmm0
mulps %xmm1, %xmm0
subps .LC52(%rip), %xmm0
mulps %xmm1, %xmm0
addps .LC53(%rip), %xmm0
mulps %xmm1, %xmm0
subps .LC54(%rip), %xmm0
mulps %xmm1, %xmm0
addps .LC55(%rip), %xmm0
mulps %xmm1, %xmm0
subps .LC56(%rip), %xmm0
```

Room for improvement by interleaving two iterations?

This is evaluated in 37 cycles?

```
mulsd %xmm2, %xmm1
movapd %xmm2, %xmm3
mulsd .LC19(%rip), %xmm0
mulsd %xmm2, %xmm3
addsd .LC21(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC22(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC23(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC24(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC25(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC26(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC27(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC28(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC29(%rip), %xmm1
```

```
mulsd %xmm2, %xmm1
subsd .LC30(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC31(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC32(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC33(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC34(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC35(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC36(%rip), %xmm1
mulsd %xmm2, %xmm1
addsd .LC37(%rip), %xmm1
mulsd %xmm2, %xmm1
subsd .LC38(%rip), %xmm1
mulsd %xmm1, %xmm3
addsd %xmm2, %xmm3
addsd %xmm3, %xmm0
ret
```

A metatrigpi

- One more parameter: compute sincos(πx), or sincos($\pi/2x$)
  - (sinpi more expensive: there is a multiplication in the argument reduction that may overflow for one value of the exponent)
- otherwise no need to add anything to the framework for this function.
- Let us go see the code?
- For generic processor, then for Kalray (partner of the ANR funding)

Conclusion

Helps collaborating with Kalray and CERN

"One of our potential customers needs sine and cos functions"

- Suggest sincospi
- Then sincospio2
- One day to get the metalibm implementation, with exhaustive test etc

A few weeks later, "One of our potential customers needs a sin/cos accurate to 15 bits"
This is now obtained by changing a parameter.

Similar efficiency on a top-secret project with CERN.

A framework based on rewriting steps?

- we accumulated some experience
- ... but the world is not ready yet
- software engineering for productivity, not for the sake of it

Looking beyond

Tarte à la crème

Study the implementation of vector CR functions

But first

Re-generate CRLibm. Who will manage that?

An obvious thing to add to the project

ATLAS-like autotuning, for when we can't predict gcc

A big issue: articulate Metalibm/C11 and Metalibm/OpenEnded

Anything to share beyond polynomial evaluation?

Thanks for your attention