Code generators for mathematical functions

Nicolas Brunie
Kalray
Grenoble, France,
nicolas.brunie@kalray.eu

Florent de Dinechin
Université de Lyon, INRIA,
INSA-Lyon, CITI,
F-69621 Villeurbanne, France
florent.de-dinechin@insa-lyon.fr

Olga Kupriianova, Christoph Lauter
Sorbonne Universités, UPMC Univ Paris 06,
UMR 7606, LIP6,
F-75005 Paris, France
{olga.kupriianova,christoph.lauter}@lip6.fr

***Abstract*—A typical floating-point environment includes support for a small set of about 30 mathematical functions such as exponential, logarithm, trigonometric and hyperbolic functions. These functions are provided by mathematical software libraries (`libm`), typically in IEEE754 single, double and quad precision.**

This article suggests to replace this `libm` paradigm by a more general approach: the on-demand generation of numerical function code, on arbitrary domains and with arbitrary accuracies.

First, such code generation opens up the `libm` function space available to programmers. It may capture a much wider set of functions, and may capture even standard functions on non-standard domains and accuracy/performance points.

Second, writing `libm` code requires fine-tuned instruction selection and scheduling for performance, and sophisticated floating-point techniques for accuracy. Automating this task through code generation improves confidence in the code while enabling better design space exploration, and therefore better time to market, even for the `libm` functions.

This article discusses the new challenges of this paradigm shift, and presents the current state of open-source function code generators available on http://www.metalibm.org/.

***Index Terms*—elementary functions, floating-point, range reduction, polynomial evaluation, libm, code generator, metalibm.**

I. Introduction & motivation

A. Standard mathematical libraries

The standard mathematical library (`libm`) provides, in a small set of precisions (single and double precision), a small set of mathematical functions:

- *elementary* functions [1] such as exponential and logarithm, sine, cosine, tangent and their inverses, hyperbolic functions and their inverses;
- and other "special" functions useful in various computing contexts, such as the power function x^y, erf, Γ or the Airy function [2].

Strictly speaking, the `libm` is specified in the C language standard (currently ISO/IEC 9899:2011). Among the other languages, some (C++, Fortran, Python) use the same library, and some redefine it in a more or less compatible way. The 2008 revision of the IEEE 754 floating-point (FP) standard [3] has attempted a common standardization. In addition, language standards also specify elementary functions on complex numbers, but for historical reasons not in the "mathematical library" section. Also, many operating systems or libraries offer, for convenience, more functions than strictly required by the `libm`. In this work we understand the term "`libm`" in its widest sense and address all these functions, and more.

B. A matter of performance

Performance of `libm` functions is of critical importance, in particular in scientific and financial computing. For example, profiling the SPICE electronic simulator shows that it spends most of its time in the evaluation of elementary functions [4]. The same holds for large-scale simulation and analysis code run at CERN [5]–[7].

The problem is that the optimal implementation of each function is dependent on the technology. Mathematical support is provided by a combination of hardware and software, and the optimal repartition between hardware and software has evolved with time [8]. For instance, the first 80287 mathematical coprocessor, in 1985, included support for a range of elementary functions (albeit in microcode) in addition to the basic operations. It was later found that software could outperform such microcode [9]. For instance, as memory got cheaper, large tables of precomputed values [10], [11] could be used to speed-up the computation of a function. Furthermore, progress in compiler technology allowed for a deeper integration of elementary functions software in the overall compilation process [12]–[14], another case for software-based functions. Today, even the relevance of hardware division and square root is disputed. At the same time, the table-based algorithms of the 90s are being replaced with high-degree polynomials [13] that behave better in the context of current highly parallel, memory-constrained multicores.

C. Limits of the library approach

1) Productivity of `libm` development: Writing a `libm` requires fine-tuned instruction selection and scheduling for performance, and sophisticated FP techniques for accuracy. Re-optimization of these mutually dependent goals for each new processor is a time-consuming and error-prone task. Besides, it is desirable that each function comes in several variants corresponding to a range of constraints on performance (e.g. optimized for throughput or optimized for latency) or accuracy. Some processor and system manufacturers (Intel, AMD, ARM, NVIDIA, HP, Apple) therefore employ teams of engineers dedicated to `libm` maintenance.

With more limited manpower, the open-source mathematical libraries (most notably in the GNU `glibc` [15] and in Newlib[1]) lag behind in performance, all the more as they have to support

[1] https://sourceware.org/newlib/

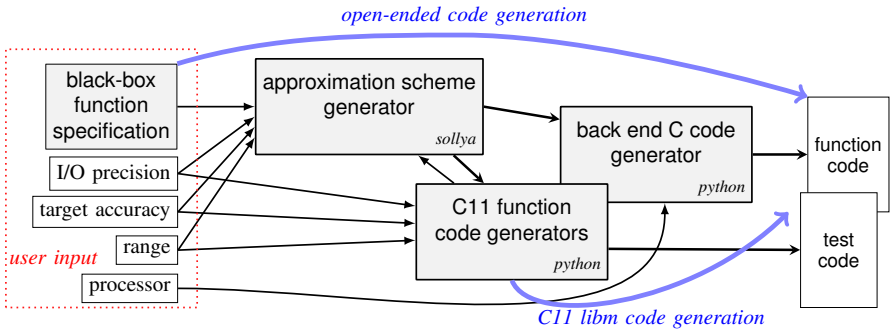

Figure 1. The proposed Metalibm tool suite

a much wider range of processors. However, the most accurate implementations are found in open-source efforts, with several correctly rounded functions provided by libraries such as IBM LibUltim [16] (now in the `glibc`) and CRLibm [17].

2) Versatility of `libm` use: Another problem is that the `libm` functions do not match all the needs of users. First, the choice is limited. Only a fraction of the functions used by CERN simulations [2] is offered by the standard `libm`. The rest is programmed when needed, sometimes without the expertise of `libm` developers. This leads to duplicated efforts and, often, poorer quality (performance- or accuracy-wise) than `libm` function.

Second, in the specific context of an application, the functions in the `libm` are often overkill. A generic library must specified to be as good as possible: argument range is specified "as far as the input and output formats allow" and accuracy "as accurate as the output format allows". However, if a programmer knows for instance that the input to a cosine will remain in a period, and that about three decimal digits of accuracy will be enough considering the accuracy of the inputs, the `libm cosf` function is too accurate (7 decimal digits) on too large a range.

The present article claims that the solution to both previous issues (productivity, and versatility) is to automate `libm` development so that functions can be generated on-demand for a wider set of contexts. This solution has already proven effective for other classes of mathematical software, most notably the ATLAS [18] project for linear algebra, and the FFTW [19] and SPIRAL [20] projects for fast Fourier transforms.

D. Use cases for generators of mathematical functions

We are currently focusing on two main use cases where we need a `libm` generator or *metalibm*, both illustrated by Figure 1.

The first one targets the widest audience of programmers. It is a push-button approach that will try to generate code on a given domain and for a given precision for an arbitrary univariate function with continuous derivatives up to some order. The function may be provided as a mathematical expression, or even as an external library that is used as a black box. We call this approach the open-ended approach, in the sense that the function that can be input to the generator is arbitrary – which does not mean that the generator will always succeed in handling it. Section III will describe how this generator has evolved from simple polynomial approximations to the generation of more sophisticated evaluations schemes, including attempts to range reduction. Here, the criterion of success is that the generated code is better than whatever other approach the programmer would have to use (composition of `libm` function, numerical integration if the function is defined by an integral, etc). "Better" may mean faster, or more accurate, or better behaved in corner cases, etc.

Still, the `libm` is here to stay, and we also want to address the needs of `libm` developments. Although the techniques presented in Section III can eventually be extended to capture all the functions of C11, it is currently not the case. There is a lot of human expertise that we are not yet able to automate. In particular, bivariate functions as `atan2` or `pow`, and some special functions, are currently out of reach. The second use case focuses on assisting people who have this expertise, not yet on replacing them. It targets a much narrower audience of programmers, those in charge of providing the actual `libm` functionality to an operating system or compiler. Here the criterion of success is that the generated code is of comparable quality to hand-written code, but obtained much faster.

The chosen approach, reviewed in Section IV, consists in giving to the `libm` developer the keys to the back end. We thus aim at offering a development framework in which the relevant evaluation schemes may be described. The challenge here is to find a proper balance between two conflicting goals: 1/ This framework should be able to capture every trick that `libm` developers use, otherwise they will not adopt it. 2/ It should nevertheless raise the level of abstraction, so that a single description generates code for the variety of code flavors and targets we want to address. And it should be fully scriptable to enable design-space exploration, but always under full control of the `libm` developer. Section IV reviews a technical solution that matches these goals.

This second use case can be viewed as a pragmatic, bottom-up approach, where we embed existing hand-crafted code in a framework to make it more generic. The first, open-ended

[2] http://project-mathlibs.web.cern.ch/project-mathlibs/mathTable.html

use case is more ambitious, more high-level, and top-down from the most abstract mathematical description of a function. These two approaches do meet as illustrated on Figure 1, and we do not claim that there is a clear border between them. For instance, the `libm` developer working directly with the back end will nevertheless invoke the evaluation scheme generator, to solve sub-problems, typically after range reduction.

II. Background on elementary function implementation

In order to understand the challenge of writing a code generator for mathematical functions, it is useful to know how they are implemented manually. The hardware provides FP additions, multiplications, bit-level manipulations, comparisons, and memory for precomputed tables. A generic technique exploiting all this is to approximate a function by tables and (possibly piecewise) polynomials.

A. Polynomial approximation

There are many ways [1] to compute an approximation polynomial p for a given function f. For our purpose, the best choice is in general a variant [21] of Remez algorithm that computes the minimax polynomial on a given interval I, *i.e.* the polynomial minimizing the approximation error

$$\varepsilon_{\text{appr}} = \|f-p\|_\infty^I = \max_{x\in I} |f-p|$$

among all the polynomials of a given degree d with FP coefficients of a given format.

For a given f and I, the larger the degree d, the better the approximation (the smaller $\varepsilon_{\text{appr}}$). When implementing a `libm`-like mathematical function, we have an upper bound constraint on $\varepsilon_{\text{appr}}$, and we look for the smallest d that satisfies this bound.

B. Argument reduction

When it is not possible to compute a polynomial of a small enough degree, the implementation domain I may be reduced. There are two basic techniques for this: split the domain into smaller subdomains, or use specific properties of the function. In both cases the approximation problem is reduced to approximating a (possibly different) function f_r on a smaller domain.

1) Domain splitting: The main idea here is to partition I into subdomains, so that on each of them we can find a minimax polynomial approximation of small degree. The splitting can be uniform, logarithmic [22], or arbitrary [23]. In the two first cases, the polynomial is selected based on a few bits from the input x. In the last case, several `if-else` statements must be used.

2) Function-specific argument reduction: For some elementary functions, specific algorithms of argument reduction may be applied [1], [10], [24]. They are based on mathematical properties such as $b^x \cdot b^y = b^{x+y}$. For example, the exponential function e^x can be computed as follows [10]. First, e^x is rewritten as

$$e^x \;=\; 2^E \cdot 2^{k-E} \cdot 2^{x\log_2 e - k} \quad .$$

Given a parameter $w \in \mathbb{N}$, we may compute $k = \lfloor 2^w x \log_2 e \rfloor 2^{-w}$ and $E = \lfloor k \rfloor$. Thus, $E \in \mathbb{Z}$ (it will provide the exponent of the result) and $k \in 2^{-w}\mathbb{Z}$. We may then compute $r = x\log_2 e - k$, a small value. Finally the exponential is rewritten as $e^x = 2^E 2^{k-E} e^r$, where the values of 2^{k-E} may be read in a precomputed table of size 2^w (indexed by k), while e^r may be well approximated by a polynomial, possibly with additional splitting.

C. Existing `libm` development tools

A mathematical function generator must build code with guaranteed accuracy. For this, it can rely on several tools that have already been used in manual `libm` development.

Sollya [25] is a numerical toolbox for `libm` developers with a focus on safe FP computations. In particular it provides state-of-the-art polynomial approximation [21], safe algorithms to compute $\varepsilon_{\text{approx}} = \|f-p\|_\infty^I$ [26] as well as a scripting language.

Gappa [17] is a formal proof assistant that is able to manage the accumulation of rounding errors in most of `libm` codes. Compared to [17], in the present work the Gappa proof scripts are not written by hand, but generated along with the C code. Interestingly, Gappa is itself a code generator (it generates Coq or HOL formal proofs).

III. Approximation of black-box functions

A. Overview

This section describes an open-source code generator written in Sollya. It inputs a parametrization file and produces corresponding C code. The parametrization file includes the function f, its implementation domain I, the desired accuracy $\bar{\varepsilon}$, maximum degree d_{max} for approximation polynomials, etc.

The function may be specified as a mathematical expression, or an external library. However, an important feature of the approach is that it doesn't require an explicit formula for the function. The proposed tool only accesses the function as a numerical black box. All the tool requires from this black box is to be able to return (in acceptable time) an arbitrarily accurate numerical evaluation of the function and its first few derivatives. This way, the approach works for functions described not only as closed-form expressions, but also as integrals, inverse functions, etc.

This black-box encapsulation of the function will not prevent the tool to exploit the function-specific range reductions presented in Section II-B2. The following shows that the required mathematical properties can be detected and exploited from a black-box function.

The tool also supports higher-than-double accuracy, using double-double and triple-double arithmetic.

B. Different levels of approximation schemes

We can structure function generation process in open-ended generator in three levels.

a) First level (polynomial approximation): This level approximates a function f by a polynomial p with an error bounded by $\bar{\varepsilon}$, and generates C code and Gappa proof.

b) Second level (piece-wise polynomial approximation): This level uses domain splitting and piecewise polynomial approximation. More details can be found in [23].

c) Third level (exploiting algebraic properties): This level attempts to detect mathematical properties, and uses them to generate argument reduction code. Properties that are currently detected by the tool include

$$f(x+y) = f(x)f(y)$$
$$f(x+C) = f(x)$$
$$f(x)+f(y) = f(xy)$$
$$f(x) = f(-x); f(x) = -f(x)$$

They respectively correspond to exponential functions $b^{x+y} = b^x b^y$, periodic functions, e.g. $\sin(x+2\pi k) = \sin(x)$, logarithms $\log_b(x)+\log_b(y) = \log_b(xy)$ and symmetrical functions (both even and odd). When such a property is detected, the tool uses the corresponding argument reduction. Then it needs an approximation scheme in the reduced domain. For this it performs a recursive call to the first or second level.

As the tool only accesses the function as a black box, it is unable to prove that the property is actually true. However, all it needs is to ensure that it is true up to some accuracy (the accuracy needed for the range reduction to work). This can be done purely numerically.

As an example, let us show how to detect the property $f(x+y) = f(x)f(y)$, which corresponds to a family of exponential functions $b^{x+y} = b^x b^y$ for some unknown base $b \in \mathbb{R}$.

First, two different points x and y are chosen in I, and the tool checks if there exists $|\varepsilon| < \bar{\varepsilon}$ such that $f(x+y) = f(x)f(y)(1+\varepsilon)$. If not, the property is not true. If yes, the tool deduces a candidate $b = \exp\left(\frac{\ln(f(x))}{x}\right)$ for some random $x \in I$. Only in this case it checks that the property is true up to the required accuracy, by computing

$$\tilde{\varepsilon} = \left\| \frac{b^x}{f(x)} - 1 \right\|_\infty^I$$

and checking if $\tilde{\varepsilon} \leq \bar{\varepsilon}$.

Other properties can be detected in the same way. For instance, periodic functions may be detected with a numerical zero search for C such that $f(x_0+C) - f(x_0) = 0$, followed by a computation of $\|f(x+C) - f(x)\|_\infty^I$.

However, handling the error bound $\tilde{\varepsilon}$ requires some analysis for certain function properties.

C. Reconstruction

The final step in mathematical functions implementation is reconstruction, i.e. the sequence of operators (or just a formula) needed to be executed to evaluate the function at some point x from the initial domain I. The reconstruction may be tricky if the generation was done on the second level and the produced code has to be vectorizable. To make the reconstruction vectorizable, some mapping function that returns an index of the corresponding interval has to be exhibited [27].

D. Several examples of code generation

The two first examples correspond to $f = \exp$ with the target accuracy $\bar{\varepsilon} = 2^{-53}$, polynomial degree is bounded by $d_{max} = 9$. Then we test a toy composite function and a sigmoid function as used in neural networks.

1) For $f = \exp$ on the small domain $I = [0, 0.3]$, the first level is enough. The generated code only consists of polynomial coefficients and polynomial evaluation function. Function generator does not handle special cases for the moment (infinities, NaNs, overflows, etc.), so for larger domains this filtering has to be added manually, this is addressed in Section IV. This function flavor is about 1.5 times faster than the standard `exp` function from the `glibc libm`.

2) For exponential flavors on larger domains, the table-driven argument reduction mentioned in Section II is used. We enlarge the domain from the previous example to $I = [0, 5]$ and set $w = 4$ for table (the table size is 2^w). On the third level, the family of exponential functions is detected. The domain is reduced to $[-\log(2)/2^{w+1}, \log(2)/2^{w+1}]$, where the first level generates polynomial code. The obtained code for this flavor executes in 10 to 60 machine cycles, with most inputs requiring less than 25 cycles. For comparison, `libm` code requires 15 to 35 cycles.

3) For the composite function $f(x) = \sin(\cos(x/3) - 1)$ on the domain $I = [-1, 1]$ with 48 bits of accuracy, using the standard `libm` will involve two function calls (both with special cases handling) and a division. In the generated code, the composite function is directly approximated by polynomials. Even though the target error is less than the mantissa length for double precision, the generated code wins both in accuracy and performance. Due to cancellation at zero, the error for composed `libm` functions explodes (Figure 3) while the error of generated function stays bounded (Figure 2). Execution of the generated code takes between 8 and 470 machine cycles, while execution of the `libm`'s analog of this flavor takes more than 150 cycles and may reach 650 cycles in the worst case.

4) Another example is the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ on the domain $I = [-2; 2]$ with 52 correct bits. No algebraic property is detected, so the generation is done on the second level. The generated code and the `libm`'s code are both of comparable accuracy and performance: execution takes between 25 and 250 cycles with most cases done within 50 cycles. The polynomial degree for the generation is bounded by $d_{max} = 9$, the domain was split to 22 subintervals.

E. Open-ended code generation: wrap up and outlook

As shown with the previous examples, and with its own 10300 lines of code, open-ended code generation has reached a certain level of maturity. It is able to quickly generate code for functions defined by various means, including black-box definitions. The produced codes do not yet reach the performance manual implementation would eventually reach but they are available quickly, at reduced cost. Final accuracy is bounded by construction and in the case of composite functions better than for plain `libm` function composition.

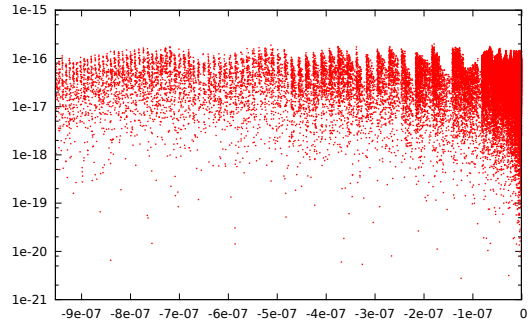

Figure 2. Logarithmic relative error of generated code on domain $[-2^{-20};-2^{-30}]$ for $\sin(\cos(x/3)-1)$

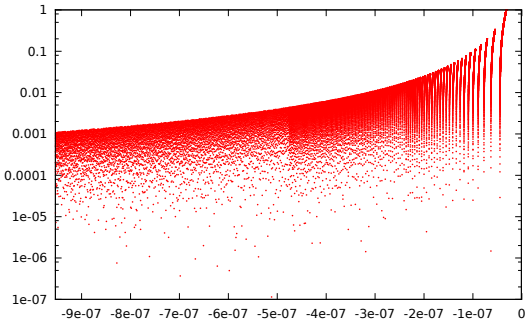

Figure 3. Logarithmic relative error of the `libm` call on domain $[-2^{-20};-2^{-30}]$ for $\sin(\cos(x/3)-1)$

However, the current prototype has some limitations that we intend to lift. First of all, unless an argument reduction can be detected, code generation is currently limited to small domains. Domain splitting can offset this limitation, but only up to some point. For instance, splitting domains like $[1;\infty]$ or just $[1;2^{1024}]$ is hopeless. A possible way to address this limitation is to transform f over a large domain $I=[a;b], a \geq 1$ into $f(1/x)$ over $[1/b;1/a]$. However, black-box static error analysis for this case remains to be studied.

Another room for improvement is the back end code generation, currently limited to generic C code. This may be addressed by the framework that we present now.

IV. A CODE GENERATION FRAMEWORK ASSISTING LIBM DEVELOPERS

We now describe a development framework that enable `libm` developers to address the productivity issue. This framework could be used to implement the previous techniques and manage the back end code generation. However, we mostly present it from the point of view of a developer of classical `libm`. As already mentionned, there is no clear boundary between the two approaches depicted on Figure 1.

Due to space restrictions, we essentially present and motivate various technical choices. A reader interested in more details is invited to look at (and experiment with) the open-source code available from www.metalibm.org.

A. General overview

All the framework is implemented in Python, a language chosen to ensure that the framework is fully scriptable. More importantly, an evaluation scheme is itself described in Python. Technically, one first defines a Python variable as being the output of the generated code, then all computations that eventually affect this variable are considered as belonging to the evaluation scheme.

A similar mechanism enables the designer to embed, in the same Python code, the analysis and proof of numerical properties of the evaluation scheme. First, intervals may be described and manipulated directly in Python using interval arithmetic. This is useful to script range and error analysis. The fact that interval computations are scripted in Python is a non-automatic, but practical way out of the correlation issues that plague naive interval arithmetic. Second, pure mathematical expression may also be described, still in the same classical Python syntax. They can be used for describing what the code is supposed to compute, so that Sollya may compute $\varepsilon_{\text{approx}}$, and Gappa may bound the accumulation of rounding errors [17].

This may seem a lot of overloading for the Python syntax. Below all this, we still also keep standard Python to script the code generation, in several steps (Figure 4).

First, the execution of the Python code describing the evaluation scheme builds a corresponding abstract syntax tree (AST). Variables and operations in the AST may be annotated with all sorts of information, either explicitly by the programmer, or automatically by the framework. Example of annotations include: desired precision, intervals for ranges and errors, probability of taking a branch, exactness of operations (no rounding errors), dependency to FP environment (rounding mode, exception visibility).

Then, the AST may be manipulated from within the same Python code. The framework provides many optimization steps for this. Some of them are similar to those implemented in generic compilers (for instance if-conversion or instruction selection). Some others are specific to the `libm` context (for instance labelling the tree with range and error intervals, or reparenthesing of arithmetic expressions to express parallelism, similarly to the CGPE approach [28]). It is also possible to write function-specific optimization steps.

An important feature is that these optimizations steps are

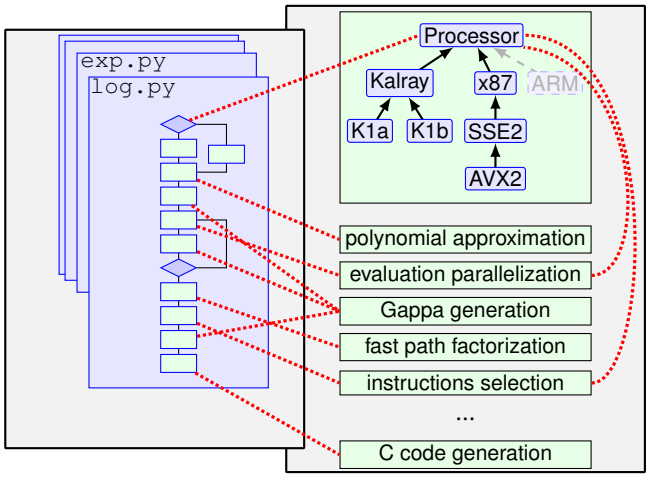

Figure 4. Zoom on the back end C code generator

invoked from the same Python code where the evaluation scheme is described. The `libm` designer may therefore chose to apply them or not (depending on the context), and decide on which part of the AST an optimization step is applied. This provides much finer control than the optimization switches of a compiler, while remaining easy to manage because elementary function codes are small. It is therefore a practical way out of the issue of conflicting compiler optimizations.

Among these optimization steps, instruction selection transforms the AST into a form closer to the generated code. It is dependent on the target hardware, as depicted in next section. The generated C is often very close to assembly: each line of C roughly matches one machine instruction, sometimes explicitly thanks to intrinsics if the target decided so. However we still leave a lot to the compiler: the detailed scheduling of the instructions, the register allocation, and the auto-vectorization.

It is in general a challenge to avoid reinventing the compiler. We want to rely on existing compilers for what they are good at. As compilers improve, some of the technical choices made here will be reevaluated.

B. The processor class hierarchy

Modern processors, even within ISA families such as IA32 or ARM, exhibit a lot of variety in their instruction sets. They differ in the basic arithmetic support (presence or not of a fused multiply-and-add, of a divide instruction, of various int/float conversion instructions). They also differ in the model of parallelism and in the capabilities of the hardware to extract this parallelism (some are VLIW, some are superscalar, some support vector parallelism; all are pipelined but the depth and width of the pipeline vary greatly). Finally, they are all increasingly memory-starved, and may offer different strategies to address this issue (caches, memory prefetching, memory access coalescing, etc). Generating optimized elementary function code for such a variety of processors is very challenging. Actually, we want to optimize the code not for a processor, but for a context that includes the processor. For instance, for the same processor, throughput-oriented or latency-oriented code may be very different.

For this, we define for each processor a class that provides information to optimization steps, and also provides code generation services. As recent processors tend to inherit the instruction set of their ancestors, the inheritance mechanism of object-oriented languages works well for overriding old techniques with newer ones when new instructions appear.

The current prototype generates code for four targets from two very different families. The first is a recent x86 processor, an out-of-order superscalar architecture. We consider an SSE2-enabled version (using the modern SIMD FPU that complements the legacy x87 unit), and a more recent AVX2-enabled one (adding hardware support for the FMA). The second family is the K1 core, developped by Kalray for its MPPA manycore processor. It implements a 5-issue VLIW in-order architecture, and comes in two versions: The first, K1A, offers a mixed single precision / double precision FP unit. The second version, K1B, adds binary64 FMA capability and two-way SIMD capabilities for binary32.

There is also a default processor target where no asumption is made on the hardware support, except IEEE-754 compliance. Support for the ARM family will be added soon.

Figure 5 shows one example of code generation service provided by the processor class: the rounding of a FP number to an int.

Python generic code:

```
k = NearestInteger(unround_k, precision = self.precision)
```

NearestInteger is a method of the generic Processor class, and generates the following code for binary32:

```
k = rintf(unround_k);
```

When overloaded in the Kalray processor class, it generates the following code:

```
t = __builtin_k1_fixed(_K1_FPU_NEAREST_EVEN, unround_k, 0);
k = __builtin_k1_float(_K1_FPU_NEAREST_EVEN, t, 0);
```

The same Python also generates binary64 versions, here for for x86 with SSE2:

```
t = _mm_set_sd(unround_k);
t1 = _mm_round_sd(t, t, _MM_FROUND_TO_NEAREST_INT);
k = _mm_cvtsd_f64(t1);
```

Figure 5. Examples of processor-specific code generation

Table I
SPEEDUPS OBTAINED WITH RESPECT TO DEFAULT LIBM.

processor	function	speedup	default `libm`
K1a	expf (binary32)	4.0	newlib
	logf (binary32)	2.7	
	exp (binary64)	5.8	
	log (binary64)	5.8	
K1b	expf (binary32)	4.0	
	logf (binary32)	2.7	
	exp (binary64)	1.8	
	log (binary64)	2.2	
core i7, SSE2	expf (binary32)	1.7	glibc
	logf (binary32)	1.02	
	exp (binary64)	1.7	
	log (binary64)	1.6	
core i7, AVX2	expf (binary32)	1.1	
	logf (binary32)	0.96	
	exp (binary64)	1.9	
	log (binary64)	1.6	

All the code variants tested in this table are C11-compliant and optimized for latency. They are generated from the same two files `exp.py` and `log.py`.

C. Some optimizations performed on the abstract syntax tree

1) Instruction selection: Instruction selection is a code generation service provided by the processor class. We could, in principle, delegate it to the compiler that does it well. To illustrate why we want to keep control on it, consider a simple example: fusing one addition and one multiplication into an FMA. This is desirable most of the time, as it improves both accuracy and latency. However, the very fact that it is more accurate impacts the error computation. The code generator therefore needs to know what FMA fusion will be performed. The impact may be deep. Consider for instance the classical Cody and Waite argument reduction [1], that computes $r = (x - kc_h) - kc_l$ where k is an integer chosen such that the first subtraction cancels. Without an FMA, we need to keep k a small integer, on a few bits, and define c_h as having so many trailing zeroes in its mantissa, so that kc_h may be computed exactly. The subtraction is then Sterbenz-exact. With an FMA,

this computation remains exact for much larger k (as long as k may be represented exactly as FP), and there is no need for zero bits in c_h, which provides more accuracy in $c_h + c_l$. Here the FMA has an impact not only on the accuracy of the result, but on the relevance domain of the range reduction.

Similar arguments apply to other instruction selection situations, such as float/int conversions.

Another interesting example is the fast reciprocal approximation. This instruction, offered by some processors to bootstrap FMA-based division, can also be used in an efficient range reduction for the logarithm [13]. In this case, it must be emulated on processors that do not offer it.

In the near future, we intend to explore using SIMD parallelism to speed-up polynomial evaluation. This requires specific instruction to move data within a SIMD vector that are extremely processor-dependent, and actually constrain the evaluation scheme. This is another case for embedding instruction selection in our code generator framework.

2) Control path generation and vectorization: Modern compilers are more and more able to auto-vectorize long, computationally intensive loops. When there are `libm` calls in such loops, these functions must be vectorized themselves.

One approach is to define vector types (which may be exposed to the programmer, but also infered by the compiler), and to provide vector versions of each `libm` function. This solution is offered for instance by Intel and AMD, at considerable development cost. Note that no standard exists (yet) for functions with vector arguments.

Another approach is to write the `libm` code in such a way that it will auto-vectorize well. Then, the vectorization itself (i.e. the use of vector instructions) is delegated to the compiler. For this to work, the code must obey certain rules, essentially regarding the tests: tests that translate to branches in the code should be avoided, and replaced with tests that translate to data selection [29]. This entails speculative execution, which typically increases the latency, as both branches of the test must be executed before the selection. However vectorization makes up for this by improving the throughput. Experiments with the Cephes library [6] showed this approach to be extremely effective, both in terms of computational efficiency and portability, with recent versions of the GCC compiler.

This second approach is therefore the one chosen here. To this purpose, there are optimization steps that specifically rework the control path of the function, depending on its intended use. For high-throughput, vectorizable code with speculative execution must be generated. For low latency code, the main objective is to make the common case fast [30], while speculative execution may still be used to exploit the ILP offered by the processor.

Initial experiments with our generator show that autovectorized code on 4-way SSE2 SIMD achieves speedups above 3 over the scalar version. This is consistent with [6].

D. A class for `libm` function

If the function is not a black box, but one of the C11 `libm` functions, it comes with a more complete specification, for instance including the management of exceptional numbers in input or output, a significant amount of the function code in a classical `libm`. This is not only a constraint, as it also enables us to use well-known techniques for a specific function.

Another advantage of having an explicit reference to the function concerns its testing. It is possible to generate random tests out of a black-box function, but we can do better if the function is known. It is possible, for instance, to design corner case or regression tests (for instance to check that a function overflows exactly when it should or handle subnormals properly – the Gappa-based proof generation does not cover special values). Also, for random testing, the default random generator can be overriden with a function-specific one that will stress the function on the parts of its domain where it is most useful. This applies to functional testing, but even more to performance testing. For instance, what matters to users of `exp` is its average performance on the small domain where it is defined, not on the full FP range where it mostly returns 0 or $+\infty$. Similarly, performance tests for `log` should use a pseudo-random number generator strongly disbalanced towards positive numbers. For `sin` it should concentrate on the first periods, but avoid over-testing the very small numbers (which represent almost half the FP numbers) where $\sin x \approx x$. Etc.

The proposed framework therefore defines a class for a C11 function with its test infrastructure. Instances of this class (one per function) may overload some methods of this class, such as the generator of regression tests (the defaults tests numbers such as the signed zeroes, infinities and NaN, and extreme threshold values of the subnormal and normal domains) and one or few generators of random input.

E. Wrapping up: current state of the project

The back end code generator (processor classes, generic optimization steps and proof generators) currently consists of 3800 lines of Python code (counted by the CLOC utility) for the open-source part covering the generic and 86-optimized processor classes. In addition there are also a few proprietary files related to the Kalray processor.

The C11 function generators currently amount to 2482 lines of code. This includes exp, log, log1p, log2, and sin/cos at various degrees of completion (they generate working code, but not always of performance comparable to handwritten code). There are also inverse, division, inverse square root, and square root implementations, essentially intended for processors without a hardware divider, such as the IA64 or Kalray. Inverse approximation is actually of more general use and can replace standard division in situations where it is needed with accuracy lower or higher than the FP precision.

A function generator typically consists of 200-300 lines of code. Considering that the generated code is about 100-200 lines, the Gappa proof about the same size, and that one generator produces many code flavors, the code generator approach should be much easier to maintain.

Near-term future work include adding ARM processors and more functions. Medium-term future work include an OpenCL back end to target GPUs (increasingly used as FP accelerators),

more evaluation scheme optimization using CGPE [28], and the generation of correctly-rounded function flavors. An interface with the PeachPy framework [31] should also be investigated.

V. Conclusions and future work

This article discusses two approaches to addressing the challenges faced by `libm` developers. The first is automated generation of evaluation schemes, to address the large number of functions of interest. The second is a specific development framework, to address the large number of hardware targets of interest. These two approaches were developed independently, and current work focuses on integrating them more. A good case study for working on such integration is the correct rounding of elementary functions. It presents several challenges, such as function evaluation in larger-than-standard precisions or less common formats, or vectorization of a technique centered on a run-time test [16].

Ideally, we wish we could have a generator where we have a clear separation (as on Figure 1) between a front-end building an approximation scheme, and a back end implementing it on a given target technology. A challenge is that the front-end itself must often be directed by the target context. A good example of this is table-based range reduction techniques. Formally capturing this complexity is currently out of reach. Happily, it is always possible, as in tools like ATLAS, to perform an empirical exploration of the parameter space: generate several variants, compile and time then, and pick up the best. This technique could also help with out-of-order processor architectures whose behavior is too hard to model anyway.

Finally, as the codes produced by the current generators are already hardened through the use of Gappa as a formal proof tool, integration of code generation into certified code development suites such as Why, or even model checkers, could be investigated.

Acknowledgements

This work was supported by the French National Research Agency through the INS program *MetaLibm*, and by an Intel donation.

The authors are thankful to Marius Cornea, Danilo Piparo, Carlos O'Donell, Vincenzo Innocente, Sergey Maidanov, Jean-Michel Muller, Siddhesh Poyarekar, and Ben Woodard for useful and interesting discussions, and for propositions on the work topics.

References

[1] J.-M. Muller, *Elementary Functions: Algorithms and Implementation (2nd edition)*. Birkhauser, 2006.

[2] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions*. National Bureau of Standards, Washington, D.C., 1964.

[3] "IEEE standard for floating-point arithmetic," IEEE 754-2008, also ISO/IEC/IEEE 60559:2011, Aug. 2008.

[4] N. Kapre and A. DeHon, "Accelerating SPICE model-evaluation using FPGAs," *Field-Programmable Custom Computing Machines*, pp. 37–44, 2009.

[5] V. Innocente, "Floating point in experimental HEP data processing," in *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012.

[6] D. Piparo, "The VDT mathematical library," in *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012.

[7] J. Apostolakis, A. Buckley, A. Dotti, and Z. Marshall, "Final report of the ATLAS detector simulation performance assessment group," CERN/PH/SFT, CERN-LCGAPP-2010-01, 2010. [Online]. Available: http://sftweb.cern.ch/AAdocuments

[8] G. Paul and M. W. Wilson, "Should the elementary functions be incorporated into computer instruction sets?" *ACM Transactions on Mathematical Software*, vol. 2, no. 2, pp. 132–142, 1976.

[9] F. de Dinechin, C. Lauter, and J.-M. Muller, "Fast and correctly rounded logarithms in double-precision," *RAIRO - Theoretical Informatics and Applications*, vol. 41, no. 1, pp. 85–102, 4 2007.

[10] P. T. P. Tang, "Table-driven implementation of the exponential function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 15, no. 2, pp. 144–157, 1989.

[11] S. Gal and B. Bachelis, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Transactions on Mathematical Software*, vol. 17, no. 1, pp. 26–45, 1991.

[12] P. Markstein, *IA-64 and Elementary Functions: Speed and Precision*, ser. Hewlett-Packard Professional Books. Prentice Hall, 2000.

[13] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Intel Press, 2002.

[14] P. Markstein, "Accelerating sine and cosine evaluation with compiler assistance," in *16th Symposium on Computer Arithmetic*. IEEE, 2003, pp. 137–140.

[15] S. Loosemore, R. M. Stallman *et al.*, *The GNU C Library Reference Manual*. Free Software Foundation, Inc.

[16] A. Ziv, "Fast evaluation of elementary mathematical functions with correctly rounded last bit," *ACM Transactions on Mathematical Software*, vol. 17, no. 3, pp. 410–423, 1991.

[17] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Transactions on Computers*, vol. 60, no. 2, pp. 242–253, Feb. 2011.

[18] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, Feb. 2005.

[19] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[20] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232– 275, 2005.

[21] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞- approximations," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 169–176.

[22] D.-U. Lee, P. Cheung, W. Luk, and J. Villasenor, "Hierarchical segmentation schemes for function evaluation," *IEEE Transactions on VLSI Systems*, vol. 17, no. 1, 2009.

[23] O. Kupriianova and C. Q. Lauter, "A domain splitting algorithm for the mathematical functions code generator," in *2014 Asilomar Conference on Signals, Systems and Computers*, 2014.

[24] P. T. P. Tang, "Table-driven implementation of the logarithm function in IEEE floating-point arithmetic," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 378 – 400, 1990.

[25] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *International Conference on Mathematical Software*, ser. LNCS, vol. 6327. Springer, Sep. 2010, pp. 28–31.

[26] S. Chevillard, M. Joldes, and C. Q. Lauter, "Certified and fast computation of supremum norms of approximation errors," in *19th Symposium on Computer Arithmetic*. IEEE, 2009, pp. 169–176.

[27] O. Kupriianova and C. Lauter, "Replacing branches by polynomials in vectorizable elementary functions," in *Book of abstracts for 16th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2014.

[28] C. Mouilleron and G. Revy, "Automatic Generation of Fast and Certified Code for Polynomial Evaluation," in *20th Symposium on Computer Arithmetic* , Aug. 2011, pp. 233–242.

[29] M. Dukhan and R. Vuduc, "Methods for high-throughput computation of elementary functions," in *Parallel Processing and Applied Mathematics*, ser. LNCS 8384, 2014.

[30] C. S. Anderson, S. Story, and N. Astafiev, "Accurate math functions on the Intel IA-32 architecture: A performance-driven design," in *7th Conference on Real Numbers and Computers*, 2006, pp. 93–105.

[31] M. Dukhan, "PeachPy: A Python framework for developing high-performance assembly kernels," in *Python for High Performance and Scientific Computing*, 2013.